Controller synthesis using interval methods

On the synthesis of non-linear state feedback controllers using interval methods

P. Kootstra

Master of Science Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

Controller synthesis using interval methods

On the synthesis of non-linear state feedback controllers using interval methods

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

P. Kootstra

May 2014

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology





Copyright © 2014, by Peter Kootstra
All rights reserved.

Abstract

This thesis investigates whether interval methods can be employed in the construction of a novel controller synthesis algorithm based on backward induction.

Interval methods are methods employing interval arithmetic, which is an arithmetic defined on real-valued intervals rather than on real-valued numbers. In the controller synthesis algorithm presented in this thesis interval methods are used to determine pre-images, represent approximations of closed sets, implement operations on these sets, and solve non-linear constrained optimisation problems without the need for derivatives.

While interval methods only impose modest requirements, i.e., they require that *interval extensions* of the difference equation describing (or approximating) the plant dynamics, cost function, and inequality constraints can be constructed, they do however suffer from the curse of dimensionality. In the presented synthesis algorithm the curse of dimensionality limits practical use to systems for which the number of states and control inputs are relatively low.

The thesis can be divided into four parts:

- The first part of this thesis (Chapters 2, 3 and 4) introduces interval arithmetic, a number of interval methods, and set computation.
- In the second part of the thesis (Chapter 5) the controller synthesis algorithm is presented and implemented using the concepts presented in the first part of the thesis.
- In the third part of the thesis (Chapter 6) the implemented synthesis algorithm is successfully used to generate, and test the viability of, controllers for two benchmark problems.
- The fourth part (Chapter 7) concludes the thesis, gives recommendations for improving the synthesis algorithm and suggests a number of topics worth considering for future research.

In conclusion, this thesis shows that interval methods can be used to construct a controller synthesis algorithm for non-trivial control problems.

Table of Contents

Acknowledgements **vii**

1 Introduction **1**

1-1 General introduction . . . 1

1-2 Problem statement . . . 2

1-3 Objectives . . . 2

1-4 Thesis outline . . . 2

1-5 Contributions . . . 3

2 Introduction to interval arithmetic **5**

2-1 Basic concepts and operations . . . 5

2-2 Interval extensions . . . 6

2-2-1 The natural interval extension . . . 7

2-2-2 The Taylor interval extension . . . 7

2-3 Drawbacks . . . 9

2-3-1 Loss of dependency . . . 9

2-3-2 The wrapping effect . . . 11

2-3-3 Divide and conquer . . . 11

2-4 Machine implementation . . . 12

2-5 Summary . . . 13

3 Interval arithmetic algorithms **15**

3-1 The Moore-Skelboe algorithm . . . 15

3-2 The set inversion via interval analysis algorithm . . . 18

3-3 Summary . . . 22

4 Set computation **23**

4-1 Recursive implementation of the SIVIA algorithm . . . 24
4-2 Point location . . . 29
4-3 Inclusion and inequality tests . . . 30
4-4 Set operations . . . 31
4-4-1 Unions . . . 31
4-4-2 Intersections . . . 32
4-4-3 Complement . . . 32
4-4-4 Projections . . . 33
4-5 Summary . . . 38

5 Controller synthesis **39**

5-1 The controller synthesis problem . . . 39
5-2 Idealised solution . . . 41
5-3 Approximate solution using interval methods . . . 46
5-3-1 Data structure . . . 47
5-3-2 Initialisation . . . 48
5-3-3 Controller synthesis algorithm . . . 49
5-3-4 Set inversion . . . 55
5-3-5 Membership test *and* value function . . . 55
5-3-6 Set projection . . . 56
5-3-7 Optimisation . . . 58
5-3-8 Set union . . . 66
5-4 Summary . . . 70

6 Benchmark problems **71**

6-1 Pendulum swing-up and stabilisation benchmark problem . . . 72
6-1-1 Modelling . . . 72
6-1-2 Control goals . . . 76
6-1-3 Controller synthesis . . . 76
6-1-4 Closed-loop system . . . 80
6-2 Car on the hill benchmark problem . . . 83
6-2-1 Modelling . . . 83
6-2-2 Control goals . . . 88
6-2-3 Controller synthesis . . . 88
6-2-4 Closed-loop system . . . 90
6-3 Summary . . . 92

7 Conclusions and recommendations **93**
7-1 Conclusions 93
7-2 Recommendations 95
7-2-1 Efficiency 95
7-2-2 Problem specific optimisations 97
7-2-3 Application specific operator implementations 99
7-3 Topics for future research 100

Bibliography **101**

Glossary **107**
List of Acronyms 107
List of Symbols 107

Acknowledgements

I would like to express my gratitude towards a number of people who have supported me during the writing of this thesis.

First, I would like to thank my family, and especially my parents, for their continued support, and making it possible for me to get this far, even when life puts up seemingly insurmountable obstacles.

Secondly, I would like to express my deepest gratitude to my supervisor, prof. dr. ir. Bart De Schutter, for his invaluable guidance, feedback, and support.

Delft, University of Technology P. Kootstra
May 2014

"If thou findest anything wanting, I shall be glad that what I have written gives thee any desire that I should have gone further. If it seems too much to thee, thou must blame the subject; for when I put pen to paper, I thought all I should have to say on this matter would have been contained in one sheet of paper; but the further I went the larger prospect I had; new discoveries led me still on, and so it grew insensibly to the bulk it now appears in. I will not deny, but possibly it might be reduced to a narrower compass than it is, and that some parts of it might be contracted, the way it has been writ in, by catches, and many long intervals of interruption, being apt to cause some repetitions. But to confess the truth, I am now too lazy, or too busy, to make it shorter."

— *John Locke*

Chapter 1

Introduction

1-1 General introduction

This thesis introduces a novel solution the problem of synthesizing state-feedback controllers for the regulation of non-linear processes. The distinguishing feature of the synthesis method described is its use of interval methods. Interval methods (algorithms employing interval arithmetic) enable the use of backwards induction to incrementally generate a regulator for a predefined region of the state space. The advantage of adopting interval methods is that the limitations imposed on the plant model used during the synthesis are greatly reduced compared with existing deterministic controller synthesis methods.

The synthesis algorithm works by constructing a sequence of backwards reachable sets of states. It starts by calculating the set of states that can reach the target states in one time-step of discrete-time model. The backward reachable states are calculated by solving a *set-inversion problem* to determine the *joint range* of the states and controller parameters that can reach the target states followed by a projection of the joint range of the states and controller parameters onto the state space. Once the set of backwards reachable states has been calculated, the controller parameters that minimise the cost-to-go for these states are determined using a *branch and bound* method.

This process is repeated while merging the controller parameters and costs with those of previous iterations until the optimal controller parameters for all backwards reachable states in the predefined region of the state space have been determined. The end result is a controller that regulates all backward-reachable states to the specified set of target states while minimising the specified cost functions.

1-2 Problem statement

Interval arithmetic computations [34] have the property that they provide *rigorous*[1] upper and lower bounds on *all* solutions of a function with a single evaluation.This property allows interval arithmetic to be used to construct a number of algorithms that have unique properties, for example: The Moore-Skelboe and related algorithms [43] *solve non-linear global optimisation* problems (for real-valued objective functions) and the Set Inversion Via Interval Analysis (SIVIA) algorithm solves *set inversion* problems[2].

One of the limiting factors for the application of these techniques is the fact that they suffer from *the curse of dimensionality*.

This thesis investigates whether interval methods can be used in the construction of a controller synthesis algorithm that retains most of the positive characteristics of interval methods while minimising the impact of their negative characteristics.

1-3 Objectives

The objective of this thesis is to present a novel general-purpose controller synthesis algorithm employing interval methods for the synthesis of (near-optimal) controllers for non-linear processes subject to (potentially) non-linear constraints and cost functions, implement this algorithm, and asses its viability by means of the creation of a number of controllers for benchmark problems.

1-4 Thesis outline

Chapter 2 introduces *interval arithmetic*, an extension of real-valued arithmetic, which is used in Chapter 3 in the algorithms (the set-inversion algorithm *SIVIA* and the *Moore-Skelboe* global optimisation algorithm) that form the basis of the controller synthesis algorithm described in this thesis.

In Chapter 4 the SIVIA algorithm presented in Chapter 3 is re-implemented as a recursive algorithm that generates binary trees representing the sets resulting from the set inversion. Also a number of set operations that operate on the aforementioned binary trees are presented, namely *unions*, *intersections*, *orthogonal projections*, *complements*, and membership tests. These algorithms form the basis of the practical implementation of the controller synthesis algorithm presented in Chapter 5.

The first part of Chapter 5 introduces the idealised version of the controller synthesis algorithm, followed by the second part that describes a practical implementation of the synthesis algorithm that can approximate the idealised version of the algorithm using extended (adding cost information and controller parameters) versions of the algorithms described in Chapter 4 together with the principles of the Moore-Skelboe algorithm presented in Chapter 3.

[1] A *rigorous* upper or lower bound is a bound that is *guaranteed* not to underestimate or overestimate the true range of a function.

[2] The set inversion is the problem of finding the *preimage* of a set by a function.

In Chapter 6 the practical implementation of the synthesis algorithm is used to generate regulators for two benchmark problems, namely the 'pendulum swing-up and stabilisation' and the 'car on the hill' benchmark problems. The viability of the generated regulators is demonstrated by using them to regulate simulated continuous-time models of the benchmark problems.

Chapter 7 concludes by discussing the results of this thesis and gives a brief summary of potential improvements and extensions of the synthesis algorithm and its implementation.

1-5 Contributions

The main contributions of this thesis are:

- The description of a novel deterministic controller synthesis algorithm based on *backward induction*.
- Showing that interval methods can be used to implement and extend a number of set-computation operations that, together with a branch-and-bound optimisation approach, can form the basis of the described synthesis algorithm.
- Demonstrating the viability of the described synthesis algorithm, by first creating a practical implementation of the synthesis algorithm and demonstrating this implementation, followed by using the practical implementation to generate controllers for the *car on the hill* and *pendulum swing-up and stabilisation* benchmark problems[3], and concluding by successfully using these controllers to regulate the states of the systems in simulations of the two benchmark problems.

[3]Controllers for a number of other benchmark problems (the *bioreactor*[40], and *double tank*[29] benchmark problems) have also been generated and tested, but due to space constraints these results are not discussed in this thesis.

Chapter 2

Introduction to interval arithmetic

This chapter introduces the basic principles of Interval Arithmetic (IA) and interval analysis, and is largely derived from [34], [16], and [44]. A more in–depth discussion of the history, applications, examples, practical implementation, and research directions of IA and interval analysis can be found in the aforementioned literature.

2-1 Basic concepts and operations

IA is a generalization of real arithmetic where real-valued numbers are replaced by real-valued closed intervals. An interval $X = [a, b]$ is the set of real numbers such that $[a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$ where $a \leq b$; when $a = b$ the interval is said to be *degenerate*. Intervals will be denoted by capital letters. The left (limit inferior) and right (limit superior) bounds of an interval X will be denoted as $\underline{X}$ and $\overline{X}$ or lower and upper bound of X respectively. The set of closed intervals in $\mathbb{R}$ is denoted by $\mathbb{IR}$. The set of *extended* intervals $\mathbb{IR}^*$ is defined as $\{[a, b] \mid a \leq b,\ a, b \in \mathbb{IR} \cup \{-\infty, +\infty\}\}$. A vector containing at least one interval-valued element is referred to as an interval vector, and a matrix containing at least one interval-valued element is referred to as an interval matrix. An interval vector is commonly referred to as a *box*.

We can redefine the elementary real-valued arithmetic operations in such a way that they can be applied to real-valued intervals. For the intervals $X = [\underline{X}, \overline{X}]$ and $Y = [\underline{Y}, \overline{Y}]$, and for the operator $\circ \in \{+,\ -,\ \cdot,\ /\}$ we can then write:

$$X \circ Y = \{[\inf x \circ y,\ \sup x \circ y] : x \in X, y \in Y\} \tag{2-1}$$

By definition an interval operation must produce a new (extended) *interval* that *contains* all possible results that can obtained by evaluating the original real-valued operation for any real-valued elements of the argument intervals. Or to quote the definition from [34]:

> An interval operation must produce a new interval containing all the possible results that can be obtained by performing the operation in question on any element, or pair of elements, of the argument intervals.

If we apply this definition to the basic real arithmetic operators, $+$, $-$, $\cdot$, and $/$, their interval equivalents become:

$$X + Y = [\underline{X} + \underline{Y}, \overline{X} + \overline{Y}] \tag{2-2}$$
$$X - Y = [\underline{X} - \overline{Y}, \overline{X} - \underline{Y}] \tag{2-3}$$
$$X \cdot Y = \left[\min(\underline{X} \cdot \underline{Y}, \underline{X} \cdot \overline{Y}, \overline{X} \cdot \underline{Y}, \overline{X} \cdot \overline{Y}), \max(\underline{X} \cdot \underline{Y}, \underline{X} \cdot \overline{Y}, \overline{X} \cdot \underline{Y}, \overline{X} \cdot \overline{Y}) \right] \tag{2-4}$$
$$X/Y = X \cdot \left[\frac{1}{\overline{Y}}, \frac{1}{\underline{Y}} \right] \text{ if } 0 \notin Y \tag{2-5}$$

with the result of Equation 2-5 being either undefined or the extended interval $[-\infty, +\infty]$ if $0 \in Y$. These definitions imply that each of the basic IA operators can be reduced to real operations and comparisons. The interval summation and multiplication are commutative, associative, and sub-distributive. It is sub-distributive due to the fact that $X(Y + Z) \subseteq XY + XZ$.

2-2 Interval extensions

Definition 1. *The* range *of a function $f : \mathbb{R} \to \mathbb{R}$ is defined as:*

$$f(X) = \{f(x) \mid x \in X\} \tag{2-6}$$

Definition 2. *An interval extension $[f] : \mathbb{IR} \to \mathbb{IR}$ of a function $f : \mathbb{R} \to \mathbb{R}$ is defined as any function that satisfies:*

$$[f](X) \supseteq f(X) \tag{2-7}$$

Definition 3. *[46] Piece-wise monotone functions, including exponential, logarithm, rational power, absolute value, and trigonometric functions, constitute the set of standard functions.*

For these standard functions there exist well-defined interval extensions that are inclusion monotonic and have an exact range enclosure (see [46]). The construction an interval extension of a real-valued function, $f(x)$, (or operator) is dependent on the properties of the function in question. For the interval extension of a monotonic function (one that is strictly non-increasing or non-decreasing, i.e., $\frac{d}{dx} f(x) \leq 0$ or $\frac{d}{dx} f(x) \geq 0$ for all x in some interval X), evaluation of its endpoints suffice; if $f(x)$ is non-decreasing the interval extension becomes:

$$[f](X) = \left[f(\underline{X}), f(\overline{X}) \right] \tag{2-8}$$

and if $f(x)$ is non-increasing the interval extension becomes:

$$[f](X) = \left[f(\overline{X}), f(\underline{X}) \right] \tag{2-9}$$

For example:

$$f(x) = e^x \tag{2-10}$$
$$[f](X) = \left[e^{\underline{X}}, e^{\overline{X}} \right] \tag{2-11}$$

and

$$f(x) = \frac{1}{1+e^{-x}} \tag{2-12}$$

$$[f](X) = \left[\frac{1}{1+e^{-\underline{X}}}, \frac{1}{1+e^{-\overline{X}}}\right] \tag{2-13}$$

Non-monotonic functions are defined on a case by case basis. For example:

$$f(x) = \tan(x) \tag{2-14}$$

$$[f](X) = \begin{cases} [-\infty, +\infty] & \text{if } \exists n \in \mathbb{Z} \text{ such that } \pi/2 + n\pi \in X \\ \left[\tan(\underline{X}), \tan(\overline{X})\right] & \text{otherwise} \end{cases} \tag{2-15}$$

with $\mathbb{Z}$ being the set of all integers and

$$f(x) = \sin(x) \tag{2-16}$$

$$[f](X) = [\underline{Y}, \overline{Y}] \tag{2-17}$$

$$\tag{2-18}$$

with

$$\underline{Y} = \begin{cases} -1 & \text{if } \exists n \in \mathbb{Z} \text{ such that } \frac{\pi}{2}\pi + 2\pi n \in X \\ \min\left\{\sin(\underline{X}), \sin(\overline{X})\right\} & \text{otherwise} \end{cases} \tag{2-19}$$

$$\overline{Y} = \begin{cases} 1 & \text{if } \exists n \in \mathbb{Z} \text{ such that } 1\frac{1}{2}\pi + 2\pi n \in X \\ \max\left\{\sin(\underline{X}), \sin(\overline{X})\right\} & \text{otherwise} \end{cases} \tag{2-20}$$

2-2-1 The natural interval extension

Any function $f : \mathbb{R}^m \to \mathbb{R}^n$ that consists of the elementary operations (as defined in Section 2-1) and the standard functions (as defined in Definition 3) can be converted into an interval-valued function, such that if $x \in X \in \mathbb{IR}$ then $f(x) \in [f](X)$. Here $[f](X)$ is constructed by replacing the real-valued operations, and functions, by their interval counterparts. An interval function constructed in this way is referred to as the *natural interval extension* of the real-valued function.

2-2-2 The Taylor interval extension

A different option for the creation of interval extensions is to create a *Taylor inclusion function* (or *Taylor extension*) (see e.g., Section 4 of [11]). This interval extension is constructed by carrying out a Taylor series expansion of the real-valued function that has to be converted into a interval extension, typically around the midpoint of the interval that is being evaluated [16][1].

[1]Note that the order of the expansion mentioned in [16] is incorrect, since the $(n+1)$th derivative of the function is not part of the n-th order Taylor expansion but it is part of the residual.

Definition 4. *The* midpoint *of an interval is defined as:*

$$m(X) = \frac{1}{2}(\underline{X} + \overline{X}) \tag{2-21}$$

Definition 5. *The* width*, or diameter, of an interval is defined as*

$$w(X) = \overline{X} - \underline{X} \tag{2-22}$$

The Taylor inclusion function can now be expressed as:

$$[f]_{\mathrm{T}}(X) = \underbrace{\sum_{i=0}^{n} \frac{\left(X - m(X)\right)^{i}}{i!} f^{(i)}(m(X))}_{n^{\text{th}}\text{-order Taylor series expansion}} + \underbrace{\frac{\left(X - m(X)\right)^{n+1}}{(n+1)!} \left[f^{(n+1)}\right](X)}_{\text{residual}} \tag{2-23}$$

with $f^{(i)}$ being the i-th derivative of f with respect to X and the residual being the interval equivalent of the *Lagrange remainder*.

Example: If we want to obtain the Taylor inclusion function for $f(x) = (x+1) \cdot \cos(x)$ for $n = 1$, we get:

$$\begin{aligned}[f]_{\mathrm{T}}(X) =& (m(X)+1)\cos\left(m(X)\right) + \\ & \left(X - m(X)\right)\cos\left(m(X)\right) - \left(m(X)+1\right)\sin\left(m(X)\right) + \\ & \frac{\left(X - m(X)\right)^{2}}{2}\left(-2\sin(X) - (X+1)\cos(X)\right)\end{aligned} \tag{2-24}$$

If we evaluate $[f]_{\mathrm{T}}(X)$ for $X = [1, 1.1]$ we obtain $[f]_{\mathrm{T}}([1, 1.1]) = [0.9523, 1.0841]$ while evaluation of the natural interval extension, $[f](X)$, yields $[0.9071, 1.1347]$ as an answer. Since $f(x)$ is strictly monotone (decreasing) for the interval $X = [1, 1.1]$, the exact range can be calculated using endpoint evaluation: $[f]^{*}([1, 1.1]) = [0.9526, 1.0806]$. It must be noted however that when, in general, the absolute and/or relative size of X increases the over-approximation of the Taylor extension can (and probably will) get worse than the over-approximation of the natural extension. If we, for example, take (2-24), with $X = [1, 3]$, we get: $[f]_{\mathrm{T}}([1, 3]) = [-6.4731, 3.7345]$, $[f]([1, 3]) = [-3.9600, 2.1613]$, and $[f]^{*}([1, 3]) = [-3.9600, 1.0806]$.

The Taylor extension can be improved by increasing the order of the Taylor expansion, allowing the over-estimation to be reduced even further.

The increase in computations would however also increase while doing this, negating the benefits of the reduction of the over-estimation of the result. Since both the natural and Taylor interval extensions yield valid interval enclosures of the true range, their intersection must also be a valid enclosure of the true range:

$$[f]_{\mathrm{T}}(X) \supseteq [f]^{*}(X),\ [f](X) \supseteq [f]^{*}(X) \implies [f]_{\mathrm{T}}(X) \cap [f](X) \supseteq [f]^{*}(X) \tag{2-25}$$

This property can be used to construct a hybrid interval extension using both the Taylor and the natural extension (or any other available extension), resulting in an enclosure that is at least as good as the best of the two extensions and potentially better.

An example of the use of a hybrid interval extension is given the example of the Moore-Skelboe algorithm, given in Section 3-1, where it is used to construct the interval extension of a cost function.

When an interval extension of a real-valued function is used in this chapter, it is implied to be a natural interval extension

If we could evaluate a function, $f : \mathbb{R}^m \to \mathbb{R}^n$ for each point $x \in \mathbb{R}^m$ in a set $X \in \mathbb{IR}^m$, we would obtain the set $f(X) \subseteq \mathbb{R}^n$. In order to approximate this set with an interval we define the *interval-range* of a function f over an interval X as:

$$[f]^*(X) = \begin{bmatrix} \left[\inf_{x\in X} f_1(x), \sup_{x\in X} f_1(x)\right] \\ \vdots \\ \left[\inf_{x\in X} f_n(x), \sup_{x\in X} f_n(x)\right] \end{bmatrix} \tag{2-26}$$

One could see $[f]^*(X)$ as the smallest box that could enclose $f(X)$.

Obtaining the exact infinum and supremum of $f(X)$ is, in general, not a tractable problem, but one can obtain an outer approximation of the range, by for example, using the divide and conquer approach that will be described in Section 2-3-3 or the Taylor interval extension described in Section 2-2-2. These approximations can in principle be arbitrarily precise, only limited by machine precision and allowed computational complexity.

It must be noted that the (true) range of a function differs from the range of an interval resulting from the evaluation of an interval extension, in that the range is guaranteed to be included within interval obtained from the evaluation, i.e., $[f]^*(X) \subseteq [f](X)$, this is due issues described in Section 2-3. The computation of the true range of a function is in general, not tractable, but can be approximated arbitrarily close using interval analysis methods.

Examples of the range of $f(X)$, the *interval-range* of $f(X)$, $[f]^*(X)$, and the result of the evaluation of an interval extension of f for X, $[f](X)$, are given in Figure 2-1.

2-3 Drawbacks

While IA has the obvious advantage of being able to carry out computations with intervals, this does not come without drawbacks. In the following sections two of these drawbacks, namely *loss of dependency*, and the *wrapping effect*, will be further investigated. Also a way of dealing with these drawbacks, using a divide and conquer approach, will be presented.

2-3-1 Loss of dependency

One of the drawbacks of IA is the so-called *loss of dependency* (also referred to as *variable disassociation*, *decorrelation*, or, *interval dependency*). This property is caused by different behaviour of IA, compared with real arithmetic, when dealing with multiple occurrences of

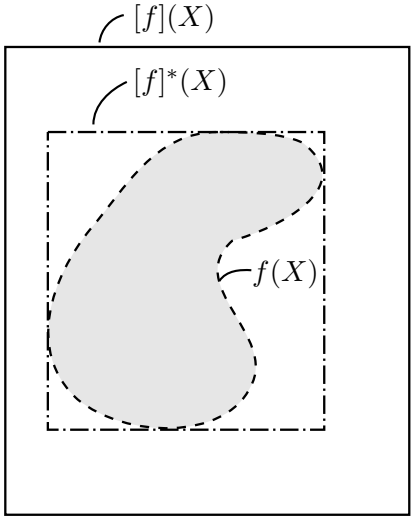


Figure 2-1: Examples of the range of $f(X)$, the interval-range, $[f]^*(X)$, and the result of evaluating $[f](X)$.

the same variable in expression. For example, the real-valued function $f(x) = (x+2)(x-2)$ can be expressed as:

$$f_a(x) = x^2 - 4, \text{expression with only a single occurrence of } x \tag{2-27}$$
$$f_b(x) = x \cdot x - 4, \text{with the square written as a multiplication} \tag{2-28}$$
$$f_c(x) = (x+2)(x-2), \text{the original, bracketed definition} \tag{2-29}$$

If we evaluate the (natural) interval extensions of these expressions for the interval $X = [-1, 2]$, with $X \in \mathbb{IR}$, we get:

$$[f_a]([-1,2]) = [0,4] - 4 = [-4,0] \tag{2-30}$$
$$[f_b]([-1,2]) = [-1,2] \cdot [-1,2] - 4 = [-2,4] - 4 = [-6,0] \tag{2-31}$$
$$[f_c]([-1,2]) = \big([-1,2] + 2\big)\big([-1,2] - 2\big) = [1,4] \cdot [-3,0] = [-12,0] \tag{2-32}$$

while all evaluations are correct (i.e. $f(X) \subseteq [f_i](X),\ i \in \{a,b,c\}$), only $[f_a]$ does not overestimate the true range of $f(X)$. This is due to the fact that in $[f_b]$ and $[f_c]$ the variable x occurred twice.

As shown in the evaluation of $[f_b]$ and $[f_c]$, ignoring the dependency leads to an overestimation of the true range of $f(X)$. One way of dealing with this issue is, when possible, to rewrite the expression so as to minimise the occurrence of the same variable. While it is generally not possible to eliminate multiple occurrences entirely, any reduction is likely to be beneficial, by both reducing the overestimation, as well as by reducing the number of computations.

2-3-2 The wrapping effect

If a function $f : \mathbb{R}^n \to \mathbb{R}^m$ is evaluated for all points in a box X, and if the result of this evaluation, $f(X)$, is not a box, then the evaluation of the interval extension of f, with box $X \in \mathbb{IR}^n$, $[f](X)$, must necessarily overestimate the range of $f(X)$, i.e., $f(X) \notin \mathbb{IR}^m \implies [f](X) \supset f(X)$. This property is called the *wrapping effect*. An illustration of the wrapping effect can be seen in Figure 2-1, where even the tightest enclosure of $f(X)$ by an interval vector, $[f]^*(X)$, is still an overestimation of the true range of $f(X)$.

2-3-3 Divide and conquer

Both the loss of dependency and the wrapping effect can be reduced by bisecting the box (dividing the box in half along one of its axes), evaluating the resulting halves, and taking the *interval hull* of the results (see e.g., Section 3 of [11]):

$$[f](X_\mathrm{L}) \sqcup [f](X_\mathrm{R}) \subseteq [f](X) \tag{2-33}$$

with boxes X_L and X_R being two halves of the box X. This procedure can be repeated for each half of the box until the desired result is achieved. The interval hull operator, $\sqcup$, is defined as:

$$[a, b] \sqcup [c, d] = \big[\min(a, c),\ \max(b, d)\big] \tag{2-34}$$

For interval vectors the interval hull operator is applied componentwise.

The approach of recursively bisecting argument boxes in order to obtain tighter bounds when evaluating an interval-valued function lies at the basis of the set inversion and the global optimisation algorithms described in Chapter 3. We can use this approach because the overestimation of the bounds of $[f](X)$ compared to the bounds of $[f]^*(X)$ is proportional to the width of X for most interval extensions [39].

The approach described in the previous paragraph will be illustrated using an example: Say we have a function $f : \mathbb{R} \to \mathbb{R}$ that can be evaluated using the expression

$$f(x) = x \cdot x \tag{2-35}$$

with its natural interval extension

$$[f](X) = X \cdot X \tag{2-36}$$

If we evaluate $[f]$ for the interval $X = [-1, 2]$, we get:

$$[f]([-1, 2]) = [-1, 2] \cdot [-1, 2] = [-2, 4] \tag{2-37}$$

which is clearly an overestimation of the true range of $f(X)$, which is $[0, 4]$. If we now divide X in two halves, $X_\mathrm{L} = [-1, 0.5]$ and $X_\mathrm{R} = [0.5, 2]$ and evaluate $[f](X_\mathrm{L})$ and $[f](X_\mathrm{R})$, and take their interval hull we get:

$$[f](X_\mathrm{L}) = [-1, 0.5] \cdot [-1, 0.5] = [-0.5, 1] \tag{2-38}$$

$$[f](X_\mathrm{R}) = [0.5, 2] \cdot [0.5, 2] = [0.25, 4] \tag{2-39}$$

$$[f](X_\mathrm{L}) \sqcup [f](X_\mathrm{R}) = [-0.5, 1] \sqcup [0.25, 4] = [-0.5, 4] \tag{2-40}$$

which is an improvement over the result of (2-37). If we now also divide X_{L} further in two halves, $X_{\mathrm{LL}} = [-1, -0.25]$ and $X_{\mathrm{LR}} = [-0.25, 0.5]$ and evaluate $[f]$ for these halves and take the interval hull of the results, we get:

$$[f](X_{\mathrm{LL}}) = [-1, -0.25] \cdot [-1, -0.25] = [0.0625, 1] \tag{2-41}$$

$$[f](X_{\mathrm{LR}}) = [-0.25, 0.5] \cdot [-0.25, 0.5] = [-0.125, 0.25] \tag{2-42}$$

$$[f](X_{\mathrm{LL}}) \sqcup [f](X_{\mathrm{LR}}) = [0.0625, 1] \sqcup [-0.125, 0.25] = [-0.125, 1] \tag{2-43}$$

which is a clear improvement on the result of (2-40). If we use the result of (2-43) instead of the result of (2-39) in (2-40) we get:

$$([f](X_{\mathrm{LL}}) \sqcup [f](X_{\mathrm{LR}})) \sqcup [f](X_{\mathrm{R}}) = [-0.125, 1] \sqcup [0.25, 4] = [-0.125, 4] \tag{2-44}$$

which is clearly an improvement on the result of (2-37).

2-4 Machine implementation

Up until this point it has been assumed that all real-valued calculations and variables are exact, i.e., performed with infinite precision. In practice computations tend to be carried out with floating point numbers, which have a limited precision. If we would implement the operations defined in Section 2-1 using floating point arithmetic instead of real arithmetic the properties defined in Section 2-1 would no longer be guaranteed to hold, i.e., an evaluation of an interval extension that does not take rounding errors into account might result in an resulting interval that does not contain all results of the evaluation of the real-valued function for all points in the argument interval. It is however still possible to guarantee these properties when using floating point numbers by using a technique called directed rounding.

The default behaviour for floating point operations, as specified in [15], is to round their results towards the nearest floating point number. While this approach minimises the rounding errors due to the precision limitations of floating point numbers it causes problems when used to implement interval operations: if we, for demonstration purposes, use a crude floating point representation that is limited to one digit after the decimal point and use it to calculate $[1.1, 1.2] \cdot 0.2$, the result would be the interval $[0.2, 0.2]$. While in practice the problem would not be as dramatic as in this example, it would still cause problems.

This problem is solved by using *directed rounding* with the interval operations. This means that the left and right bounds of the result of the operation are respectively rounded towards $-\infty$ and $+\infty$ (the floating point equivalents of *floor* and *ceil*) instead of being rounded towards the nearest floating point number. If we apply this principle to the previous example, the resulting interval would become $[0.2, 0.3]$, which would properly enclose the true interval $[0.22, 0.24]$ as specified in Section 2-1.

An important factor in the increase of the popularity of interval methods is the support of directed rounding modes in IEEE 754, the IEEE standard for floating-point arithmetic [15], which is implemented in practically all CPUs equipped with a floating-point unit built after the year 1985. This enables implementation of interval operations with minimal overhead.

2-5 Summary

In this chapter a short introduction to IA has been given, giving an overview of its implementation and some of its properties.

It has also been shown that any closed-form[2] real-valued expression can be transformed into an interval-valued expression, the so-called *interval extension* of the real-valued expression.

These interval extensions can be constructed using a number of methods. Two of these methods, namely the *natural interval extension*, and the *Taylor interval extension* have been demonstrated using a simple example. Finally it has been demonstrated how the overestimation of the used interval extensions can be reduced and how IA can be implemented using *floating point* calculations while retaining its *inclusion* properties.

[2]We define a closed-form expression as an expression that consists of a finite number of elementary functions [5].

Chapter 3

Interval arithmetic algorithms

In this chapter two algorithms using Interval Arithmetic (IA) will be introduced, the *Moore-Skelboe algorithm*[1], which is a general-purpose solver for non-linear optimization problems, and the Set Inversion Via Interval Analysis (SIVIA) algorithm, which is a general-purpose *set inversion* algorithm. Both algorithms will be presented in their most basic form. In Chapter 5 the SIVIA algorithm will be combined with the principles of the Moore-Skelboe algorithm and the *set computation* operations presented in Chapter 4 to form the basis of the controller synthesis algorithm presented in this thesis.

3-1 The Moore-Skelboe algorithm

The Moore-Skelboe algorithm was first proposed in [47] where it was used as a procedure to reduce the over-approximation of interval calculations, using a subdivision strategy similar to the one described in Section 2-3-3. A number of variations of the Moore-Skelboe algorithm exist, all having the same structure, but with different stopping criteria [51], subdivision strategies [8], or subdivision selection criteria [9].

The implementation of the Moore-Skelboe algorithm shown in Algorithm 1 finds upper and lower bounds of the value of the global minimum (or minima), within a pre-defined tolerance ϵ, and the subset of X_{range} in which, given the tolerance, the minimum (or minima) can lie. This is accomplished by dividing and evaluating the initial search box X_{range} into increasingly smaller subboxes, updating the current lowest upper bound $\tilde{y}^*$ if the evaluation of a subbox yields a lower upper bound than $\tilde{y}^*$ and discarding boxes that can be proven to be suboptimal by comparing the lower bounds of their evaluation to the current lowest upper bound.

This continues until the width of the boxes becomes less than $\epsilon/2$, at which point the boxes and the value of their evaluation are placed in the set of potential solutions L. Finally all entries in L that are suboptimal (i.e. their lower bounds exceed the current value of $\tilde{y}^*$) are removed. The boxes remaining in L have the property that the lower bound of their evaluated

[1] Note that the Moore-Skelboe algorithm itself is not explicitly used, only its underlying principles.

value, Y, is smaller than $\tilde{y}^*$ and the width of their evaluated value, $w(Y)$, is smaller or equal to $\epsilon/2$. These two properties together have as a consequence that the worst case width of the union of all Ys is equal or less than ϵ, which satisfies the requirements imposed on the solution of the solver.

The worst-case width of the cost, i.e., $w(\bigcup Y_i) = \epsilon$ will occur when $L = \{(X_1, Y_1), (X_2, Y_2))\}$, with $Y_1 = [\tilde{y}^*,\ \tilde{y}^* + \epsilon/2]$ and $Y_2 = [\tilde{y}^* - \epsilon/2,\ \tilde{y}^*]$. In this case the union of the Ys in the solution set will have its maximal width. For this union to have a width larger than ϵ it must have at least one Y with a width larger than $\epsilon/2$ or contain a Y that is guaranteed to be sub-optimal. Because these conditions cannot occur, since they are explicitly checked for in Algorithm 1, the 'true' value of the global minimum must always lie within an interval with a maximum width of ϵ.

Algorithm 1: Global optimizer using IA.

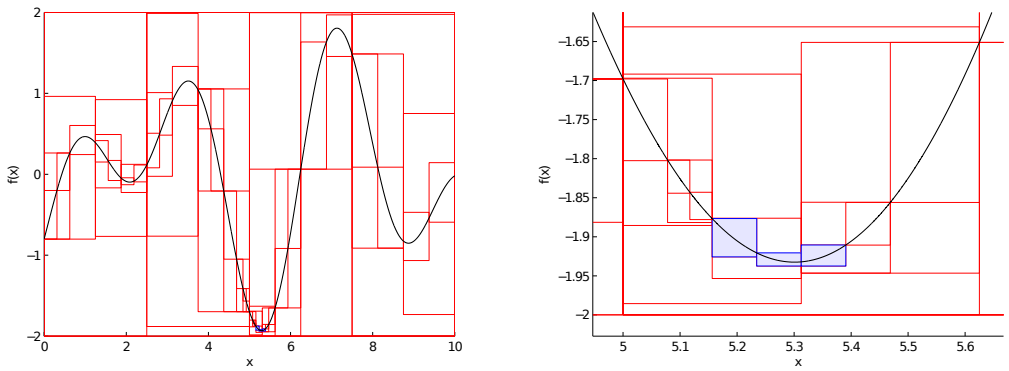
input : An interval extension, $[f] : \mathbb{IR}^n \to \mathbb{IR}$, of cost function $f : \mathbb{R}^n \to \mathbb{R}$, an initial search domain, $X_{\text{range}} \in \mathbb{IR}^n$, and a maximum width of the minimum cost ϵ.

output : A list of tuples, L, containing boxes, X, and bounds on their cost, Y, such that $\forall (X, Y) \in L$ it holds that $\overline{Y} - y^* \leq \epsilon$, with y^* being the global minimum of f on X_{range}, and an upper bound on the minimum, $\tilde{y}^*$.

variables : $L,\ M = \{\}$,
$y \in \mathbb{IR},\ \tilde{y}^* \in \mathbb{R}$.

begin

$L \longleftarrow \emptyset$
$Y \longleftarrow [f](X_{\text{range}})$
$\tilde{y}^* \longleftarrow \overline{Y}$
$M \longleftarrow \{(X_{\text{range}}, Y)\}$
while $M \neq \emptyset$ **do**
SELECT select and remove an element (X, Y) from M
if $\underline{Y} > \tilde{y}^*$ **then** `// check whether current` Y `is suboptimal`
continue `// current` Y `is suboptimal`
else
if $\overline{Y} < \tilde{y}^*$ **then**
$\tilde{y}^* \longleftarrow \overline{Y}$ `//` $\tilde{y}^*$ `is replaced by the improved upper bound`
if $w(Y) \leq \epsilon/2$ **then** `// check if` Y `is a potential solution`
$L \longleftarrow L \cup \{(X, Y)\}$
else
BISECT bisect X to obtain X_{l} and X_{r}
$Y_{\text{l}} \longleftarrow [f](X_{\text{l}})$
$Y_{\text{r}} \longleftarrow [f](X_{\text{r}})$
$M \longleftarrow M \cup \{(X_{\text{l}}, Y_{\text{l}}), (X_{\text{r}}, Y_{\text{r}})\}$
remove all tuples (X, Y) from L for which $\underline{Y} > \tilde{y}^*$
return L



(a) All boxes. **(b)** Zoom in on the solution set.

Figure 3-1: Example of the boxes evaluated by the Moore-Skelboe algorithm.

Table 3-1: Results of the Moore-Skelboe example.

X	Y
$[5.1563, 5.2344]$	$[-1.9258, -1.8767]$
$[5.2344, 5.3125]$	$[-1.9374, -1.9205]$
$[5.3125, 5.3906]$	$[-1.9374, -1.9105]$

At the line marked SELECT in Algorithm 1 a tuple (X, Y) is selected and removed from M, typically the tuple with the lowest $\overline{Y}$, since the X in this tuple has a good probability of containing the minimiser.

At the line marked BISECT the current X will be split into two sub-boxes if the current Y is neither sub-optimal ($\overline{Y} < \tilde{y}^*$) nor a potential solution ($w(Y) \leq \epsilon/2$). The split typically occurs along the widest axis of the box. Figure 3-1 gives an illustration of the subboxes of the initial search domain $X_{\text{range}} = [0, 10]$ that have been evaluated by the Moore-Skelboe algorithm to find the global minimum of the real-valued function $f(x) = \sin(2x + 0.2) + \sin(1.25x - 1.6)$ in X_{range} with $\epsilon = 0.1$. Using a hybrid interval extension of f using the intersection (i.e. $\left[\max(\underline{Y_{\text{N}}}, \underline{Y_{\text{T}}}), \min(\overline{Y_{\text{N}}}, \overline{Y_{\text{T}}})\right]$) of the result of the evaluation of the natural interval extension, Y_{N}, and the result of the evaluation of the third-order Taylor interval extension of f, Y_{T}. The evaluated boxes are shown in red, the boxes in the solution set are shown in blue. The main loop of Algorithm 1 was executed 63 times in order to obtain the solution set shown in Table 3-1. It must be noted that the reduction of the search domain by bisection alone is not recommended since significant speed-ups can be achieved by using additional reduction techniques known as *contractors* (see Chapter 4 of [16]).

A contractor is any technique, except bisection, that can reduce the size of a box. An example of a contractor that can be used if an interval extension of the first derivative of the objective function is available, is the so-called *interval Newton* algorithm [27]. Another method for speeding up the Moore-Skelboe algorithm is to carry out an additional evaluation of the real-valued cost function at the midpoint of each box and to use the result as a potential new lowest upper bound $\tilde{y}^*$ instead of using the upper bound of the current evaluation result $\overline{Y}$. If we use this method with the previous example, we reduce the number of executions of the

main loop of Algorithm 1 from 63 to 33, and even with the additional midpoint evaluations a reduction of the running time by roughly 50% is obtained while retaining the guaranteed properties of the solution of the original algorithm.

One of the main distinguishing features of the Moore-Skelboe algorithm (and of most other IA based optimisation algorithms) is that convergence to the global minima is guaranteed even when faced with discontinuities and local minima [33] [25]. This in contrast to point methods where, in general, no such guarantees can be given [12]. This is due to the way the Moore-Skelboe algorithm functions; by eliminating all parts of the potential solution set that can be proven to be sub-optimal, and doing this exhaustingly until a predefined accuracy is attained we can guarantee that what remains of the initial potential solution set must contain the global optimum (or optima) insofar it is (or they are) located within the initial potential solution set.

3-2 The set inversion via interval analysis algorithm

Definition 6. *The problem of finding the set $\Xi \subset \mathbb{R}^n$, given a function $f : \mathbb{R}^m \to \mathbb{R}^{*m}$ and a set[2] $\Psi \subseteq \mathbb{R}^*$, such that $\Psi = f(\Xi)$, i.e., find the set of arguments Ξ for which the function f maps to the set Ψ, is known as the* set inversion problem.

The SIVIA algorithm described in this section was first proposed in [17][3] and [18] and is a general solution to the *set inversion problem.*

SIVIA is a *set inversion* algorithm that, for any given real-valued function $f : \mathbb{R}^n \to \mathbb{R}^{*m}$ for which an interval extension, $[f] : \mathbb{IR}^n \to \mathbb{IR}^{*m}$, can be constructed (as described in Section 2-2) and a given set $\Psi \subseteq \mathbb{R}^{*m}$, can determine an inner (Ξ^-) and outer (Ξ^+) approximations of the set $\Xi \subseteq \mathbb{R}^n$, $\Xi^- \subset \Xi \subset \Xi^+$, that asymptotically approach Ξ such that $\Psi = f(\Xi)$ i.e., the set Ξ is the *pre-image* or *inverse image* of the set Ψ under f.

The SIVIA algorithm works by evaluating increasingly smaller subsets of the potential solution set Ξ with the interval extension $[f]$ of f; on the resulting values an *inclusion test* is carried out. We can distinguish five different cases, as illustrated in Figure 3-2. In two cases, when evaluating box X_a and X_b, we can conclude that X_a is inside the solution set, since $f](X_a) \subseteq \Psi$, and X_b is outside the solution set, since $[f](X_b) \not\subseteq \Psi$. For the remaining boxes, X_c, X_d, and X_e, it is at this time not possible to determine if they are part of the solution set or not, or intersect with both the solution set and its complement. These boxes are bisected and evaluated until either the result is a subset of Ψ or Ψ^c, or until some stopping criterium is satisfied, for example if a lower bound on the width of the evaluated box is reached. The left-hand side of Figure 3-2 shows the unknown solution set Ξ, together with five boxes for which $[f]$ is evaluated.

The right-hand side of the figure shows the set Ψ, for which we want to obtain the inverse image, and the boxes resulting from the evaluating of $[f]$ for the boxes shown on the left hand side of the picture, also the 'ideal' (i.e. what we would obtain if we could evaluate f for all points x inside the box in question) are shown. Table 3-2 sums up the set-theoretic properties of the various cases shown in Figure 3-2. The operation of determining whether

[2]The set of extended reals, $\mathbb{R}^*$, is defined as $\mathbb{R} \cup \{-\infty, +\infty\}$.

[3]In the application where SIVIA was used in [17] SIVIA was referred to as *Set Enclosure Via Interval Analysis*, SEVIA.

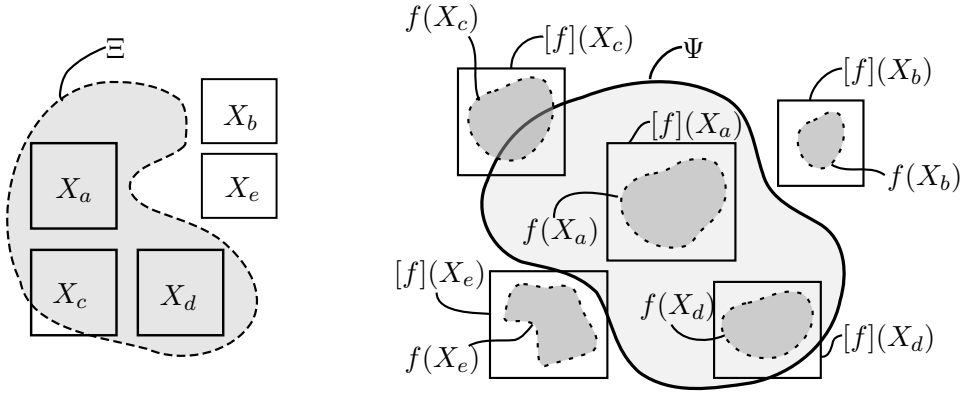


Figure 3-2: Illustration different scenarios.

result ideal evaluation	result interval evaluation
$f(X_a) \subseteq \Psi$	$[f](X_a) \subseteq \Psi$
$f(X_b) \not\subseteq \Psi$	$[f](X_b) \not\subseteq \Psi$
$f(X_c) \cap \Psi \neq \emptyset,\ f(X_c) \cap \Psi^c \neq \emptyset$	$[f](X_c) \cap \Psi \neq \emptyset,\ [f](X_c) \cap \Psi^c \neq \emptyset$
$f(X_d) \subseteq \Psi$	$[f](X_d) \cap \Psi \neq \emptyset,\ [f](X_d) \cap \Psi^c \neq \emptyset$
$f(X_e) \not\subseteq \Psi$	$[f](X_e) \cap \Psi \neq \emptyset,\ [f](X_e) \cap \Psi^c \neq \emptyset$

Table 3-2: Different scenarios for the evaluation of $[f]$.

$[f](X)$ is either a subset of Ψ or Ψ^c ($\Psi^c = \mathbb{R}^m \backslash \Psi$, or the complement of Ψ), or intersects both, is named an *inclusion test*[4]. The implementation of the inclusion test differs depending on the representation of the set Y. An example of an inclusion test on an interval vector is given in (3-4), (3-5), and (3-6) below. But for now we assume an appropriate procedure exist.

For both X_a and X_b the evaluation of $[f]$ gives a guaranteed result, i.e. the interval evaluation of X_a, and X_b, are guaranteed to be a subset of Ψ, or a subset of Ψ^c respectively, and since $f \subseteq [f]$ the results also hold for the 'ideal' evaluation. When a guaranteed result is obtained the box is placed in the Ξ^{inside} or Ξ^{outside} sets, depending on whether the boxes are subsets of Ψ or Ψ^c. For X_c, X_d, and X_e the evaluation of $[f]$ yields a result that intersects both with the set Ψ as well as its complement Ψ^c; for the 'ideal' evaluation this is true for X_c, unlike the results of the 'ideal' evaluation of X_d, and X_e, which are respectively subsets of Ψ and Ψ^c. This difference between the interval and 'ideal' evaluations is caused by over-approximation of the interval evaluation. The last three cases are handled by bisecting the boxes and evaluating both halves. This process is repeated until the evaluation of a box yields a guaranteed result, or if its width falls below a specified value. When the latter is the case, the box is added to the $\Xi^{\text{undetermined}}$ set. At some point all sub-boxes of the initial search box, Ξ_0, have been placed in either Ξ^{inside}, Ξ^{outside}, or $\Xi^{\text{undetermined}}$, at which point the algorithm terminates.

The inner approximation Ξ^- of the solution of the set inversion is identical to Ξ^{inside}, and the outer approximation, Ξ^+, consists of the union of Ξ^{inside} and $\Xi^{\text{undetermined}}$. The combination of the sets Ξ^{inside}, Ξ^{outside}, and $\Xi^{\text{undetermined}}$ is referred to as a *sub-paving*[5].

When using SIVIA with interval extensions of real-valued functions the quality of the resulting

[4]See paragraphs 2.5.2 and 2.5.3 of [16].
[5]See Chapter 3 of [16]

Algorithm 2: Set inversion via interval analysis (SIVIA).

input : An interval extension, $[f] : \mathbb{IR}^n \to \mathbb{IR}^m$, of $f : \mathbb{R}^n \to \mathbb{R}^m$, a target set $\Psi \subseteq \mathbb{IR}^m$, for which the set inverse has to be determined, and an initial search box, $\Xi_0 \in \mathbb{IR}^n$ in which the solution set is located[6].

output : The solution set, represented by three lists of non-overlapping boxes, Ξ^{inside}, $\Xi^{\text{undetermined}}$, and, Ξ^{outside}, containing the boxes that are, respectively, strictly inside the solution set, undetermined, or strictly outside the solution set.

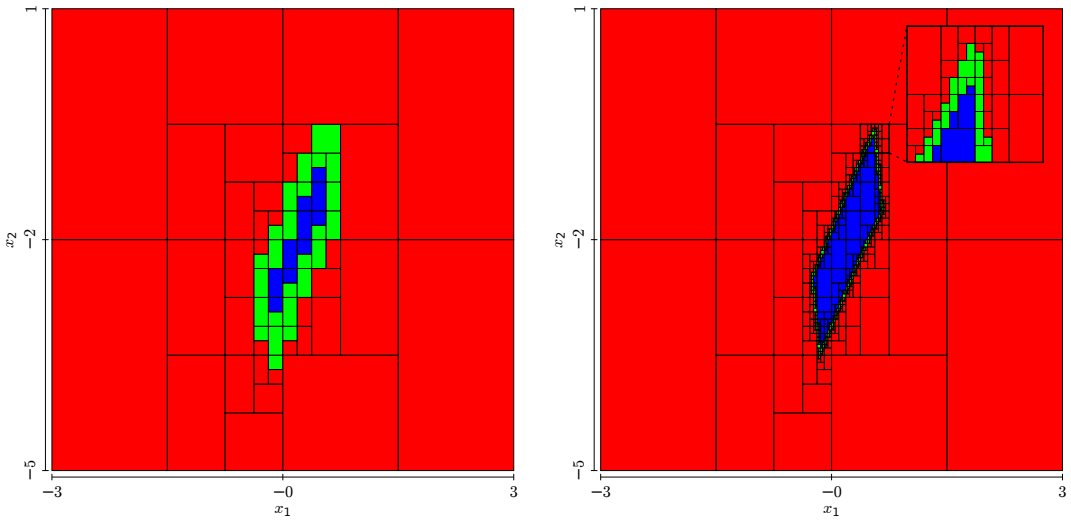
variables : Ξ^{inside}, $\Xi^{\text{undetermined}}$, Ξ^{outside}, $\Xi^{\text{undetermined}}_{\text{temp}} = \{\}$, $X, X_{\text{l}}, X_{\text{r}} \in \mathbb{IR}^n$, $Y \in \mathbb{IR}^m$

begin

- $\Xi^{\text{undetermined}}_{\text{temp}} \longleftarrow \Xi_0$
- $\Xi^{\text{inside}} \longleftarrow \emptyset$
- $\Xi^{\text{undetermined}} \longleftarrow \emptyset$
- $\Xi^{\text{outside}} \longleftarrow \emptyset$
- **while** $\Xi^{\text{undetermined}}_{\text{temp}} \neq \emptyset$ **do**
  - select and remove an element, X from set $\Xi^{\text{undetermined}}_{\text{temp}}$
  - $Y \longleftarrow [f](X)$
  - **if** $Y \subseteq \Psi^{\text{c}}$ **then** `// Y and Ψ do not intersect`
    - $\Xi^{\text{outside}} \longleftarrow \Xi^{\text{outside}} \cup \{X\}$
  - **else if** $Y \subseteq \Psi$ **then** `// Y is a subset of Ψ`
    - $\Xi^{\text{inside}} \longleftarrow \Xi^{\text{inside}} \cup \{X\}$
  - **else** `// all other cases`
    - **if** *X and/or Y satisfy the stopping criteria* **then**
      - $\Xi^{\text{undetermined}} \longleftarrow \Xi^{\text{undetermined}} \cup \{X\}$
    - **else**
      - bisect X to obtain X_{l} and X_{r}
      - $\Xi^{\text{undetermined}}_{\text{temp}} \longleftarrow \Xi^{\text{undetermined}}_{\text{temp}} \cup \{X_{\text{l}}, X_{\text{r}}\}$

inversion depends primarily on the stopping criteria used. A typical approach is to stop bisecting a box when its width falls below a specified value; if this occurs the box is placed in $\Xi^{\text{undetermined}}$. When a regular paving (which will be discussed in more detail in Chapter 4) is constructed, i.e., the bisections are done at the midpoint of the box, cycling through its axes, then the width of the sub-box is directly related to the number of bisections of the initial search box that have led to the respective sub-box of the initial search box. On the other hand, if SIVIA is implemented recursively, as will be done in Chapter 4, the width of a sub-box will be directly related to the recursion depth of the algorithm at which the sub-box is evaluated.

If we for example use SIVIA, with target set $\Psi \subset \mathbb{IR}^2$, and real-valued function $f : \mathbb{R}^2 \to \mathbb{R}^2$, to obtain an approximate set-inversion $\Xi = f^{-1}(\Psi) \cap \Xi_0$ with Ξ represented by the three sets



(a) Maximum recursion depth = 10. **(b)** Maximum recursion depth = 16.

Figure 3-3: Inversion results with a maximum recursion depth of 10 and 16.

Ξ^{inside}, $\Xi^{\text{undetermined}}$, and Ξ^{outside} with:

$$\Xi_0 = [-3.0, 3.0] \times [-5.0, 1.0] \tag{3-1}$$

$$\Psi = [-0.5, 0.5] \times [-0.5, 0.5] \tag{3-2}$$

$$f(x) = \begin{bmatrix} x_1 + 0.1x_2 \\ 2 - 1.962\sin(x_1) + 0.8x_2 \end{bmatrix} \tag{3-3}$$

and inclusion test:

$$\forall i \in \{1,2\},\ \underline{\Psi}_i \leq Y_i \leq \overline{\Psi}_i \implies X \in \Xi^{\text{inside}} \tag{3-4}$$

$$\exists i \in \{1,2\},\ \underline{\Psi}_i > \overline{Y}_i \text{ or } \overline{\Psi}_i < \underline{Y}_i \implies X \in \Xi^{\text{outside}} \tag{3-5}$$

$$\text{otherwise} \implies X \in \Xi^{\text{undetermined}} \tag{3-6}$$

with $Y = [f](X)$ and using a natural interval extension $[f] : \mathbb{IR}^2 \to \mathbb{IR}^2$ of f, at two different recursion depths, 10 and 16, we obtain the results shown in Figures 3-3a and 3-3b with Ξ^{inside} shown in blue, $\Xi^{\text{undetermined}}$ shown in green, and Ξ^{outside} shown in red.

At each recursion a single axis is bisected at its midpoint. This results in a maximum of 32 (2^5) divisions for each axis if the maximum recursion depth is 10, and 256 (2^8) divisions for each axis if the maximum recursion depth is 16. If we plot the average execution times[7] of the SIVIA algorithm, for the example, for a range of different maximum recursion depths we obtain Figure 3-4. This figure clearly shows that the execution time *increases* roughly exponentially when the maximum recursion depth is increased. This behaviour also applies to other set inversion methods, and is the one of the main limitations of the SIVIA algorithm. It must be noted, however, that the accuracy, as measured by the relative size of the $\Xi^{\text{undetermined}}$ set exhibits a roughly exponential *decrease* when the recursion depth increases. Also the computational complexity increases exponentially with the number of dimensions of the solution set [17] [18].

[7] Using a C++ implementation of the SIVIA algorithm, under Debian 6.0 on an "Oracle Virtual Box" virtual machine, on a PC with an Intel i7 930 processor at 2.8 GHz with 12 GB RAM.

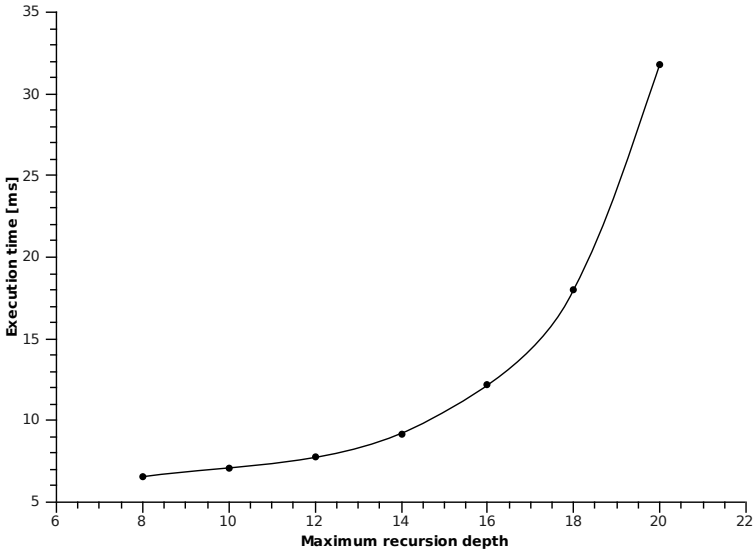


Figure 3-4: Execution times for varying maximum recursion depths.

3-3 Summary

In this chapter it has been shown that using IA it is possible to construct an optimiser that, in principle, can find the location(s) and value of the global minima with a pre-determined accuracy within a box for a given (closed-form) expression (optionally) subject to constraints.

It also has been shown how the SIVIA algorithm can compute the *inverse image* of a set inside a specified range, by systematically partitioning and evaluating the specified range.

Chapter 4

Set computation

The implementation of the SIVIA algorithm and the set operators presented in this chapter together with the principles of the Moore-Skelboe algorithm presented in Section 3-1 will be combined to form the controller synthesis algorithm presented in Chapter 5.

By themselves the algorithms presented in this chapter can be used as a general purpose set computation toolbox that is able to compute set inversions, unions, intersections, orthogonal projections, and complements.

In the previous chapter it was shown how the SIVIA algorithm can obtain inner and outer approximations of a set inversion. The output of the SIVIA algorithm, the approximation of the inverted set, Ξ, was represented by three sets, Ξ^{inside}, Ξ^{outside}, and $\Xi^{\text{undetermined}}$, each of the sets represented as the union of lists of boxes as described in Section 3-2.

The problem with representing these sets as lists of boxes is that performing operations on them is rather inefficient. For example, if we want to determine whether a given point is located within the inverted set or not, all the boxes in the three lists have to be checked until a box is found that contains the given point. This implies that in the worst case all boxes have to be checked. Since these inverted sets can potentially consist of a large number of boxes, checking them exhaustively is clearly undesired.

Also determining the intersection of two sets represented as boxes is either simple but computationally inefficient (if we determine the union of the intersection of each box of one set with all boxes of the other set), or complex but computationally efficient (if we first sort the lists of boxes, and determine the union of the intersection of the boxes that potentially intersect) [53].

In this chapter an alternative representation of the solution set of set inversions is presented: a *binary tree*. The solution set represented by this binary tree is identical to the solution set represented by the lists of boxes generated the SIVIA algorithm presented in the previous chapter. But rather than storing the boxes resulting from the bisections, the bisections themselves are stored in the form of a binary tree. In this chapter it will be shown how set operations, like unions, intersections, membership tests, and orthogonal projections that operate on sets represented using binary trees can be implemented.

The SIVIA, union, and intersection, algorithms described in this chapter are directly based on [22][1]. However the pseudo-code shown in this chapter is closer to a practical implementation of the algorithms than the pseudo-code shown in [22]. For a high-level overview of the algorithms the reader is referred to [22].

The *union* and *intersection* algorithms described in this chapter extend the algorithms described in [22] by retaining their `UNDETERMINED` labels, rather than regarding them as being labelled `OUTSIDE`[2].

The rationale for retaining the `UNDETERMINED` boxes is twofold[3]: First, the total volume of the boxes labelled `UNDETERMINED` can be used as an indicator of the quality of the inversion. Secondly, if the quality of the inversion is inadequate, it can be improved by increasing the maximum recursion depth and continuing at the boxes where SIVIA previously gave up, i.e., the boxes labelled `UNDETERMINED`.

Also, when using sets without boxes labelled `UNDETERMINED`, as is done in [22], the set operations, union, intersection, and negation, follow the rules of *boolean logic* with `INSIDE` and `OUTSIDE` respectively corresponding to *true* and *false*, when boxes labelled `UNDETERMINED` are also used the rules of *Kleene's three-valued logic* K_3 [23] are followed.

The `pavingMembershipTest` described in Algorithm 9 is similar to the INSIDE algorithm described in Section 3.3 of [22], but differs from it in that it exploits the fact that the membership test is done on a regular sub-paving in order to reduce the number of inequality checks that have to be carried out.

Section 4-4-4 describes an algorithm for determining orthogonal projections of regular sub-pavings (see Definition 8), that, to the author's best knowledge, has not yet been described before, although it must be noted that the same result as a set inversion followed by an orthogonal projection can be achieved by using a modified version of the SIVIA algorithm, as is described by the SIVIAPY algorithm in Section 5.3 of [16].

4-1 Recursive implementation of the SIVIA algorithm

In this section it will be shown how the SIVIA algorithm can be implemented as a recursive function that generates a binary tree representing the resulting set inversion.

First we define a number of functions that are used in the pseudo-code of the recursive implementation of SIVIA.

The `newnode` function creates a new binary-tree node and sets its left and right child nodes as empty.

The `freenode` function deallocates all data associated with its argument node. Also `node.left` and `node.right` are set to $\emptyset$, ensuring `isLeaf` recognises `node` as a leaf node.

The `isLeaf` function returns `TRUE` if the left and right child nodes of its argument node are empty; otherwise `TRUE` is returned.

[1] This paper consists of a large portion of Chapter 3 of [16].

[2] Instead of using the *labels* `INSIDE`, `OUTSIDE`, and `UNDETERMINED`, [22] and [16] use, respectively, the *logical values* 1, 0, and $[0, 1]$.

[3] Note that these properties are not used in the implementation of the controller synthesis algorithm described in Section 5-3, although a future version of the synthesis algorithm could certainly use these properties.

The `bisect` function shown in Algorithm 3 splits box X into two halves, along the midpoint of the i-th axis. It does this by copying X to two new boxes, X_{L} and X_{R}, and adjusting the values of their i-th axis.

For example, evaluating `bisect`$([-1,1] \times [-1,1],\ 2)$ would return $X_{\mathrm{L}} = [-1,1] \times [-1,0]$ and $X_{\mathrm{R}} = [-1,1] \times [0,1]$.

Algorithm 3: Definition of the `bisect`$(X,\ i)$ function.

input : A box $X \in \mathbb{IR}^n$, and the index, `idx` $\in \{1,\ldots,n\}$, of the axis to bisect
output : Two boxes $X_{\mathrm{L}} \in \mathbb{IR}^n$ and $X_{\mathrm{R}} \in \mathbb{IR}^n$

begin
  foreach $i \in \{1,\ldots,n\}$ **do**
    if $i =$ *idx* **then**
      $X_{\mathrm{L}}^i \longleftarrow [\underline{X}^i,\ m(X^i)]$
      $X_{\mathrm{R}}^i \longleftarrow [m(X^i),\ \overline{X}^i]$
    else
      $X_{\mathrm{L}}^i \longleftarrow X^i$
      $X_{\mathrm{R}}^i \longleftarrow X^i$
  return $(X_{\mathrm{L}}, X_{\mathrm{R}})$

In Algorithm 3 X^i is the i-th element of the interval vector X.

The `inclusiontest` function, shown in Algorithm 4, returns the set-theoretic relation between the box X and the set Ψ.

How the relation between X and Ψ is determined depends on how the set Ψ is represented. This can be an inequality check, as described in (3-4), (3-5), and (3-6) and implemented in Algorithm 10 or a membership test on the output of the SIVIA algorithm, as will be discussed in Section 4-3.

Algorithm 4: Definition of the `inclusiontest`$(X,\ \Psi)$ function.

input : A box $X \in \mathbb{IR}^n$, and a closed subset $\Psi \subset \mathbb{R}^n$
output : A return value of `INSIDE`, `OUTSIDE`, or `UNDETERMINED`

begin
  if $X \subseteq \Psi$ **then**
    return `INSIDE`
  else if $X \subseteq \Psi^{\mathrm{c}}$ **then**
    return `OUTSIDE`
  else
    return `UNDETERMINED`

The `reunite`(`node`) function, shown in Algorithm 5, replaces a node having two child nodes (that are also leaf nodes) with the same value, by a leaf node having this value. Since the `reunite` function's argument is *called by reference*, i.e., it is a pointer, the line "return `node`" can also be omitted, but explicitly returning the `node` (pointer) allows the `reunite` function to be, for example, used as: "return `reunite`(`newNode`)" in the various algorithms. Note that

in Algorithm 5, as well as in all other algorithms (like Algorithms 6, 8, 9, 11, 12, and 13) taking as an argument, or returning as a result, a (sub) tree, are *called by reference*, i.e., they take as an argument, or return as a result, a pointer to the root node of a (sub) tree.

Algorithm 5: Definition of the `reunite`(node) function.

input : A binary tree node, `node`, representing a regular sub-paving.
output : A binary tree node, representing a *minimal* regular sub-paving.

begin
  if *isLeaf(node==FALSE)* **then**
    if *isLeaf*(*node.left*) *and* *isLeaf*(*node.right*) *and* *node.left.value* = *node.right.value* **then**
      node.value $\longleftarrow$ node.left.value
      freenode(node.left)
      node.left $\longleftarrow \emptyset$
      freenode(node.right)
      node.right $\longleftarrow \emptyset$
  return node

The recursive implementation of the SIVIA algorithm itself is shown in Algorithm 6.

Algorithm 6: Definition of the `SIVIA`($[f],\ X, \Psi,$ `currDepth`, `maxDepth`) function.

input : Interval extension $[f] : \mathbb{IR}^m \to \mathbb{IR}^n$, target set $\Psi \subset \mathbb{R}^n$, search box $X \in \mathbb{IR}^m$, current recursion depth, `currDepth`, and maximum recursion depth, `maxDepth`.
output : The root node of a binary tree containing the paving approximating $f^{-1}(\Psi)$.

begin
  node $\longleftarrow$ newnode()
  Y $\longleftarrow$ $[f](X)$
  node.value $\longleftarrow$ inclusiontest(Y, Ψ)
  if *node.value* = *UNDETERMINED* *and* *currDepth* $<$ *maxDepth* **then**
    bisectionIndex $\longleftarrow$ (currDepth mod m) $+ 1$
    $(X_{\mathrm{L}}, X_{\mathrm{R}})$ $\longleftarrow$ bisect(X, bisectionIndex)
    node.left $\longleftarrow$ SIVIA($[f],\ X_{\mathrm{L}}, \Psi,$ currDepth $+ 1,$ maxDepth)
    node.right $\longleftarrow$ SIVIA($[f],\ X_{\mathrm{R}}, \Psi,$ currDepth $+ 1,$ maxDepth)
    return reunite(node)
  return node

The recursive version of the `SIVIA` algorithm shown in Algorithm 6 is, with respect to the evaluated boxes, identical to the algorithm described in Algorithm 2. The difference lies in the fact that the binary tree that is created by repeated bisections is retained instead of using a list to store the boxes. Also note that while our algorithm does not explicitly store the sub-boxes (they can always be inferred from the root-box and their position in the binary tree), a practical implementation might still explicitly store the boxes in each node, reducing the number of computations (midpoint calculations) required when traversing the tree, at the

cost of increased memory usage.

Also note that there are a number of scenarios where it is recommended that the evaluation of $[f](X)$ and the inclusion test `inclusiontest`(Y, Ψ) are combined into a single function. This is especially the case if the inclusion test is an inequality check, like Algorithm 10, and if the result of $[f](X)$ consists of a (large) number of (independent) elements. When this is the case, an integrated algorithm can immediately return `OUTSIDE` when the evaluation yields an element larger than 0, removing the need for further evaluations (they are redundant anyway).

The output of the `SIVIA` algorithm is a so-called *Regular Sub-Paving (RSP)* of X.

A sub-paving is defined as [16]:

Definition 7. *A* sub-paving *of a box $X \in \mathbb{IR}^n$ is a union of non-overlapping subboxes of X with non-zero width.*

A *regular* sub-paving is defined[4] as:

Definition 8. *A sub-paving of $X \in \mathbb{IR}^n$ is* regular *if each of its boxes can be obtained from X by a finite succession of bisections and selections.*

The set of all *regular sub-pavings* of a box $X \in \mathbb{IR}^n$ is denoted as $\mathcal{RSP}(X)$. A sub-paving of X that covers X is also referred to as a paving of X.

For the remainder of this chapter, whenever we talk about a set, it is implied that this is a set approximated by a binary tree containing a regular sub-paving.

Figure 4-1 shows the binary tree that is obtained when we use the recursive version of SIVIA to generate the paving shown in Figure 3-3a. This tree contains a total of 137 nodes: 68 intermediate, 10 labelled `INSIDE`, 34 labelled `OUTSIDE`, and 25 labelled `UNDETERMINED`.

Figure 4-2 shows the binary tree that would be obtained if the `reunite` function was not used when determining the same set-inversion. This tree contains a total of 183 nodes: 91 intermediate, 10 labelled `INSIDE`, 41 labelled `OUTSIDE`, and 41 labelled `UNDETERMINED`[5].

Note that the set sets represented by the trees shown in Figures 4-1 and 4-2 are indentical.

[4] This is a slightly modified version of definition given in Section 3.3.1 of [16].

[5] That the number of nodes labelled INSIDE and labelled OUTSIDE are both 41 is coincidental.

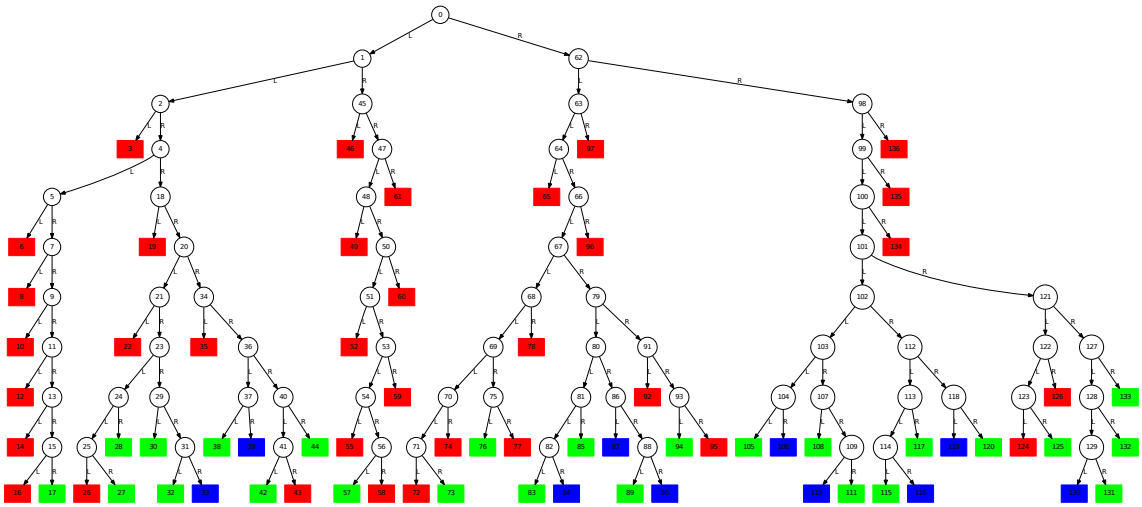


Figure 4-1: The binary tree corresponding to the paving shown in Figure 3-3a.

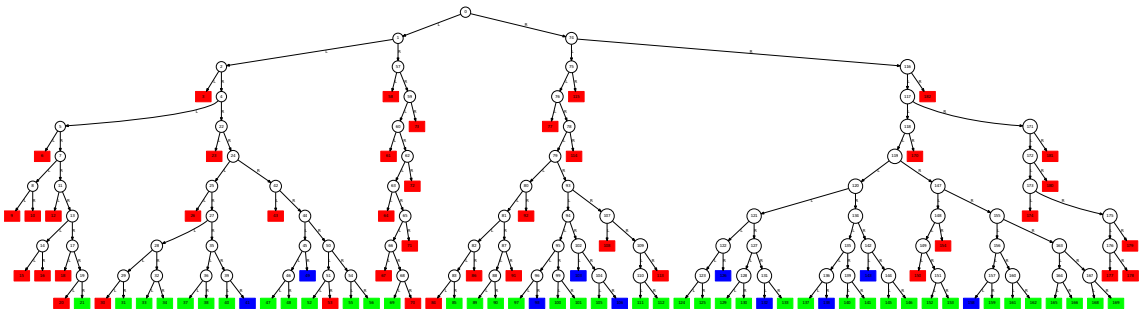


Figure 4-2: The binary tree constructed without using the `reunite` function.

4-2 Point location

The `findNode` algorithm described in Algorithm 7 accepts three arguments, the rootnode, `node` of an RSP of box, $X \in \mathbb{IR}^n$, a point, $y \in \mathbb{R}^n$, and the current recursion depth, `depth`, and it returns a[6] leaf-node of the RSP that intersects with y.

In this algorithm, and in Algorithm 9, it is assumed that each node also contains its associated box. If the nodes do not contain the box values they can be added to the tree using Algorithm 8. These boxes are used to extract the midpoint of the axis in which its the child-nodes are divided.

It is assumed that the condition $y \in X$ is checked before this algorithm is used.

When the point y coincides with both child boxes, i.e., when $y_i = \texttt{midpoint}$, the ambiguity is resolved by returning[7] `findNode`(`node.right`, y, `depth` + 1).

Algorithm 7: Definition of the `findNode`(`node`, y, `depth`) function.

input : The root-node, `node`, of a regular sub-paving of $X \in \mathbb{IR}^n$, a point, $y \in X$ and the current recursion depth, `depth`.

output : The leaf-node of the regular sub-paving, associated with a box that intersects with y.

```
begin
    if isLeaf(node) then
        return node
    i ⟵ (depth mod n) + 1
    midpoint ⟵ node.left.box_i
    if y_i < midpoint then
        return findNode(node.left, y, depth + 1)
    else
        return findNode(node.right, y, depth + 1)
```

The `addBoxValues` function shown in Algorithm 8 adds the box values to each node in the binary tree representing an RSP of a box $X \in \mathbb{IR}^n$.

Algorithm 8: Definition of the `addBoxValues`(`node`, X, `depth`, `boxDim`) function.

input : The root node of a binary tree, `node`, a box, $X \in \mathbb{IR}^{\texttt{boxDim}}$, the current recursion depth `depth`, and the dimensionally of the box X, `boxDim`.

output : The binary tree with its nodes now containing their associated boxes.

```
begin
    node.box ⟵ X
    if isLeaf(node = FALSE) then
        i ⟵ (depth mod boxDim) + 1
        (X_L, X_R) ⟵ bisect(X, i)
        addBoxValues(node.left, X_L, depth + 1, boxDim)
        addBoxValues(node.right, X_R, depth + 1, boxDim)
    return node
```

[6] In cases where the point is located at the common boundary of two child-nodes the right child node is returned.

[7] Another option might be to evaluate both branches and return the node with the 'best' properties.

4-3 Inclusion and inequality tests

In this section two algorithms are described that can be used to determine the set-theoretic relation between a box and a given set.

The `pavingMembershipTest` function shown in Algorithm 9 determines the set-theoretic relation between a box and a set, when the set is approximated using a regular sub-paving represented by a binary tree.

Algorithm 9: Definition of the `pavingMembershipTest`(`node`, X, `depth`) function.

input : The root node of an RSP, `node`, a box, $X \in \mathbb{IR}^n$, and the current recursion depth, `depth`.

output : The membership of X with respect to the set approximated by the paving with root node `node`.

begin

- **if** *isLeaf*(*node*) **then**
  - return `node.value`
- $i \longleftarrow (\texttt{depth} \bmod n) + 1$
- `midpoint` $\longleftarrow \overline{\texttt{node.left.box}}_i$
- **if** $\overline{X}_i \leq$ *midpoint* **then**
  - return `pavingMembershipTest`(`node.left`, X, `depth` $+ 1$)
- **if** $\underline{X}_i \geq$ *midpoint* **then**
  - return `pavingMembershipTest`(`node.right`, X, `depth` $+ 1$)
- `leftValue` $\longleftarrow$ `pavingMembershipTest`(`node.left`, X, `depth` $+ 1$)
- **if** *leftValue* $=$ *UNDETERMINED* **then**
  - return `UNDETERMINED`
- `rightValue` $\longleftarrow$ `pavingMembershipTest`(`node.right`, X, `depth` $+ 1$)
- **if** *rightValue* $=$ *UNDETERMINED* **then**
  - return `UNDETERMINED`
- **if** *leftValue* $=$ *INSIDE* *and* *rightValue* $=$ *INSIDE* **then**
  - return `INSIDE`
- **if** *leftValue* $=$ *OUTSIDE* *and* *rightValue* $=$ *OUTSIDE* **then**
  - return `OUTSIDE`

The `inequalityMembershipTest` function shown in Algorithm 10 implements (3-4), (3-5), and (3-6) for $\Psi = \{y \in \mathbb{R}^n \mid y_i \leq 0,\ i \in \{1, \ldots, n\}\}$.

Algorithm 10: Definition of the `inequalityMembershipTest`(X) function.

input : A box, $X \in \mathbb{IR}^n$.
output : The membership of X with respect to the set $\{y \in \mathbb{R}^n \mid y_i \leq 0,\ i \in \{1, \ldots, n\}\}$.

```
begin
    InsideFlag ⟵ TRUE
    foreach i ∈ {1, ..., n} do
        if X̲_i > 0 then
            return OUTSIDE
        else if X̄_i > 0 then
            InsideFlag ⟵ FALSE
    if InsideFlag = TRUE then
        return INSIDE
    else
        return UNDETERMINED
```

4-4 Set operations

In this section a number of set operations are described that operate on sets described by RSPs. For binary operations it is assumed that they operate on RSPs of the same box.

4-4-1 Unions

The union of two sets, $A, B \in \mathcal{RSP}(X),\ X \in \mathbb{IR}^n$ can be determined by recursively traversing the binary trees of both pavings in parallel. We do this by starting at the root-node of both trees, traversing them, in parallel, depth-first, while simultaneously constructing the output tree (the tree representing $A \cup B$). Algorithm 11 shows how this can be implemented, with the $\cup$ operator used in case (1) specified in Table 4-1.

When traversing the trees in Algorithm 11 we encounter three different cases:

- Both nodes are *not* leaf nodes - case (1): In this case the value of the left and right child nodes are determined by evaluating the `setUnion` function for the left and right child nodes of the two input trees.
- Both nodes are leaf nodes - case (2): In this case the value of the current node of the output tree is determined by Table 4-1, i.e., if at least one node has the label `INSIDE`, the output node label is set to `INSIDE`; if both labels are `OUTSIDE`, the output node label is set to `OUTSIDE`; otherwise, the output node label is set to `UNDETERMINED`.
- Only one node is a leaf node - case (3a) and (3b): This is similar to the previous case, but differs in that the leaf node is used directly as an argument. This is allowed since in this case only the value of the node is relevant.

Table 4-1: Properties of the *union* operator, $\cup$, with I=INSIDE, O=OUTSIDE, and U=UNDETERMINED

$\cup$	I	O	U
I	I	I	I
O	I	O	U
U	I	U	U

Algorithm 11: Definition of the setUnion(PavingIn1, PavingIn2) function.

input : Two pavings PavingIn1 and PavingIn2 for which the union has to be determined.
output : An RSP containing the union of PavingIn1 and PavingIn2.

```
begin
    PavingOut ⟵ newnode()
(1) if isLeaf(PavingIn1) = FALSE and isLeaf(PavingIn2) = FALSE then
        PavingOut.left ⟵ setUnion(PavingIn1.left, PavingIn2.left)
        PavingOut.right ⟵ setUnion(PavingIn1.right, PavingIn2.right)
(2) else if isLeaf(PavingIn1) = TRUE and isLeaf(PavingIn2) = TRUE then
        PavingOut.value ⟵ PavingIn1.value ∪ PavingIn2.value
        return PavingOut
    else
(3a)    if isLeaf(PavingIn1) = TRUE then
            PavingOut.left ⟵ setUnion(PavingIn1, PavingIn2.left)
            PavingOut.right ⟵ setUnion(PavingIn1, PavingIn2.right)
(3b)    else if isLeaf(PavingIn2) = TRUE then
            PavingOut.left ⟵ setUnion(PavingIn1.left, PavingIn2)
            PavingOut.right ⟵ setUnion(PavingIn1.right, PavingIn2)
    return reunite(PavingOut)
```

4-4-2 Intersections

The intersection of two sets, $A, B \in \mathcal{RSP}(X)$, with $X \in \mathbb{IR}^n$ can be determined in a similar fashion as the union of two sets. Algorithm 12 shows how the intersection operation can be implemented.

The implementation of Algorithm 12 is almost identical to that of Algorithm 11; it only differs in that in case (2) the intersection operator, $\cap$, is used instead of the union operator, $\cup$. The $\cap$ operator is specified in Table 4-2.

4-4-3 Complement

The complement, $A^C \in \mathcal{RSP}(X)$ of a set $A \in \mathcal{RSP}(X)$ can be determined by creating a copy of A and replacing all nodes labelled INSIDE with OUTSIDE, and all nodes labelled OUTSIDE with INSIDE. The nodes labelled UNDETERMINED remain labelled UNDETERMINED.

Table 4-2: Properties of the *intersection* operator, $\cap$, with I=INSIDE, O=OUTSIDE, and U=UNDETERMINED

$\cap$	I	O	U
I	I	O	U
O	O	O	O
U	U	O	U

Algorithm 12: Definition of the `setIntersection(PavingIn1, PavingIn2)` function.

input : Two pavings `PavingIn1` and `PavingIn2` for which the intersection has to be determined.

output : An RSP containing the intersection of `PavingIn1` and `PavingIn2`.

```
begin
    PavingOut ⟵ newnode()
(1) if isLeaf(PavingIn1) = TRUE and isLeaf(PavingIn2) = TRUE then
        PavingOut.left ⟵ setIntersection(PavingIn1.left, PavingIn2.left)
        PavingOut.right ⟵ setIntersection(PavingIn1.right, PavingIn2.right)
(2) else if isLeaf(PavingIn1) = TRUE and isLeaf(PavingIn2) = TRUE then
        PavingOut.value ⟵ PavingIn1.value ∩ PavingIn2.value
        return PavingOut
    else
(3a)    if isLeaf(PavingIn1) = TRUE then
            PavingOut.left ⟵ setIntersection(PavingIn1, PavingIn2.left)
            PavingOut.right ⟵ setIntersection(PavingIn1, PavingIn2.right)
(3b)    else if isLeaf(PavingIn2) = TRUE then
            PavingOut.left ⟵ setIntersection(PavingIn1.left, PavingIn2)
            PavingOut.right ⟵ setIntersection(PavingIn1.right, PavingIn2)
    return reunite(PavingOut)
```

4-4-4 Projections

The projection algorithm shown in Algorithm 13 creates a sub-paving, $\Xi_{\mathrm{P}} \in \mathcal{RSP}(X_{\mathrm{P}})$ of the orthogonal projection, *along* a specified axis, of a sub-paving $\Xi \in \mathcal{RSP}(X)$ with $X_{\mathrm{P}} \in \mathbb{IR}^{n-1}$ and $X \in \mathbb{IR}^n$.

The projection algorithm works by recursively replacing the current node by the union of its left and right child nodes when the axis of the box associated with the node, specified by `axis`, is bisected, i.e., when `axis` $= i$. This process is repeated until only leaf nodes remain. At this point the set shown in the last subfigure of Figure 4-3, which corresponds to the tree shown in Figure 4-15, is obtained. This set is the final result of the projection.

This process is illustrated in Figures 4-3 to 4-15. Figure 4-3 shows what happens to the boxes associated with the leaf nodes of the binary tree of the set that is projected, while Figures 4-4 to 4-15 show the trees themselves. The left and right sub-trees that are to be merged (by

Algorithm 13: Definition of the `projectPaving(node, axis, depth, boxDim)` function.

input : The rootnode, `node`, of an RSP, the axis, `axis`, along which the projection has to take place, the current recursion depth, `depth`, and the dimensionality, `boxDim`, of the set represented by the RSP.

output : An RSP containing the projection of the set represented by the original RSP, along the axis specified by `axis`.

```
begin
    if isLeaf(node) = TRUE then
        newNode ⟵ newnode()
        newNode.value ⟵ node.value
        return newNode
    i ⟵ (depth mod boxDim) + 1
    if axis = i then
        newNode ⟵ setUnion(node.left, node.right)
        return projectPaving(newNode, axis, depth + 1, boxDim)
    else
        newNode ⟵ newnode()
        newNode.left ⟵ projectPaving(node.left, axis, depth + 1, boxDim)
        newNode.right ⟵ projectPaving(node.right, axis, depth + 1, boxDim)
        return reunite(newNode)
```

taking their union), and their associated boxes, are marked with solid lines the sub-trees and boxes that have been obtained by the previous merger are marked with dashed lines.

After the final step of the projection (when we have obtained paving 11a of Figure 4-3) the values of the boxes are adjusted to reflect that the binary tree of the paving represents the projection. This results in paving 11b of Figure 4-3. This step is not shown in Algorithm 13 but in the practical implementation of the controller synthesis algorithm this step is carried out by Algorithm 22.

Also note that the binary trees of the intermediate stages (Figures 4-5 to 4-14) should not be regarded as representing an RSP. This due to the fact that they contain sub-trees representing sub-pavings of both 2-D and 1-D boxes at the same time.

Orthogonal projections of a set on its subspaces can be implemented by repeated application of Algorithm 13 using different indices, or by extending Algorithm 13 to accept multiple values of the `axis` argument, i.e., changing the "if `axis` $= i$" condition into "if `axis` $\in$ `indexes`" where `indexes` is a list containing the indices of the axes that have to be projected away.

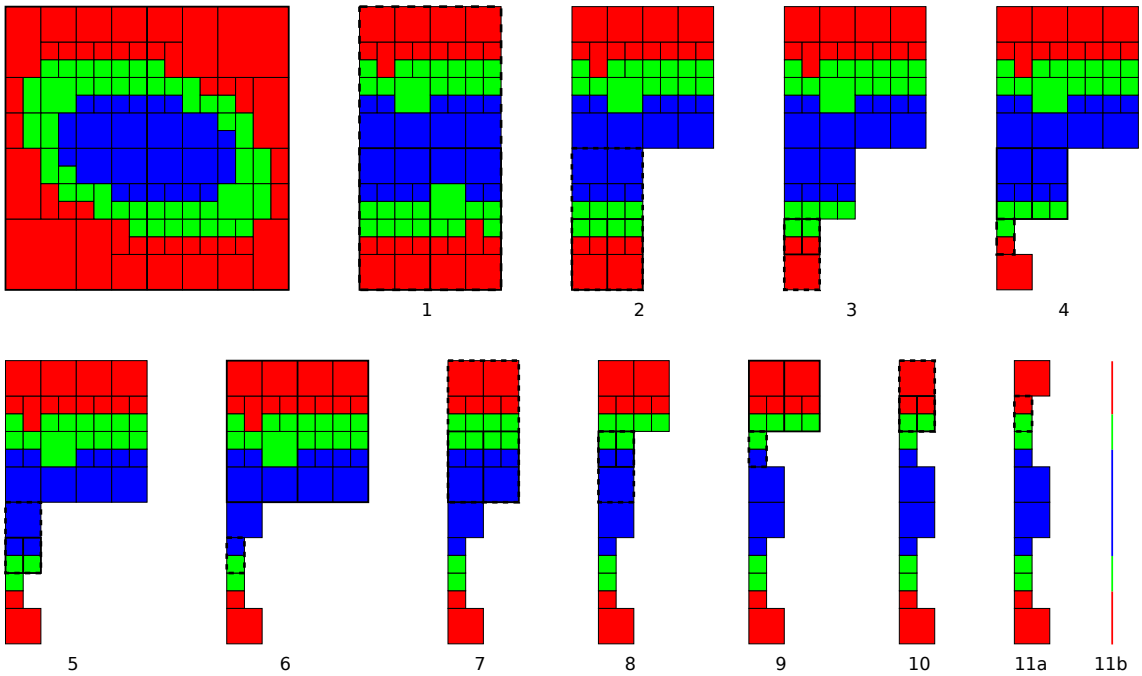


Figure 4-3: An example of the different stages of a paving during orthogonal projection on the vertical axis. With areas marked with solid and dashed lines corresponding to the equally marked sub-trees in Figures 4-4 to 4-15.

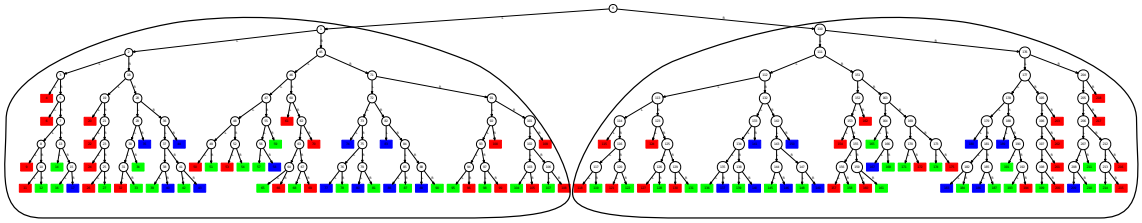


Figure 4-4: The binary tree of the original paving.

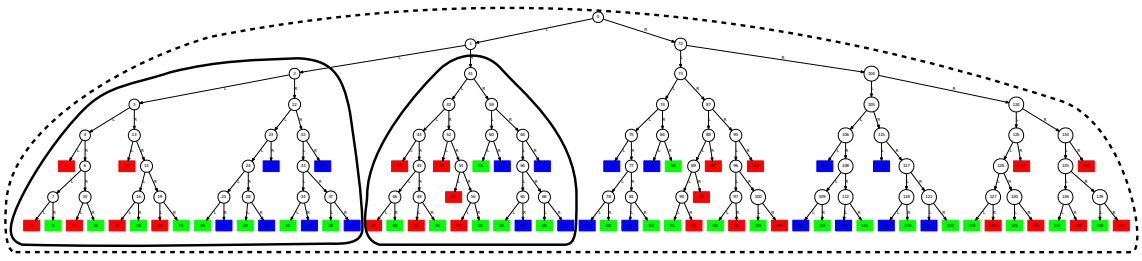


Figure 4-5: The binary tree of the first stage.

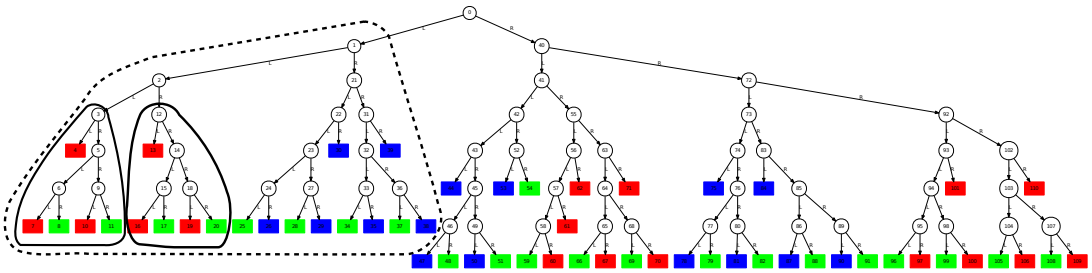


Figure 4-6: The binary tree of the second stage.

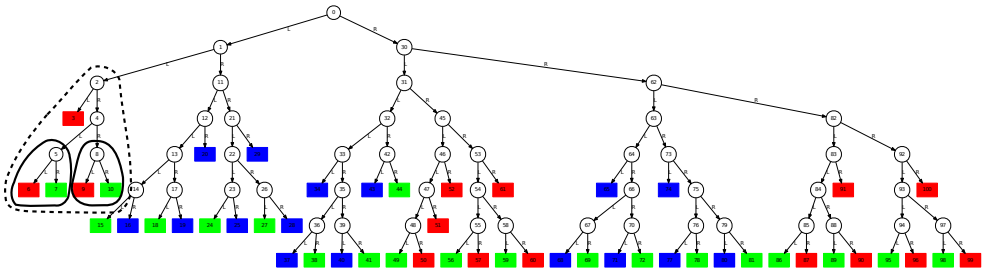


Figure 4-7: The binary tree of the third stage.

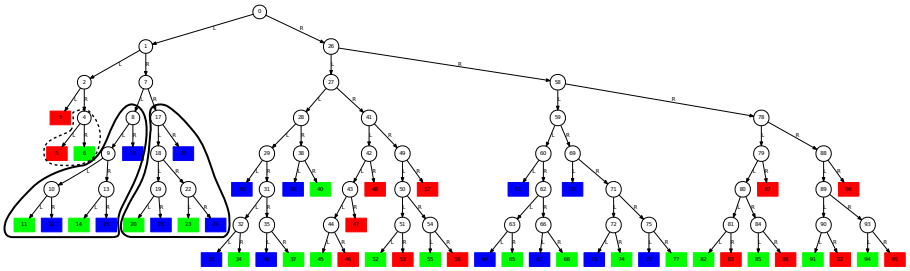


Figure 4-8: The binary tree of the fourth stage.

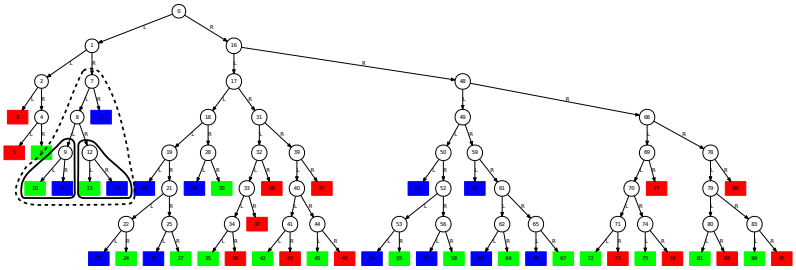


Figure 4-9: The binary tree of the fifth stage.

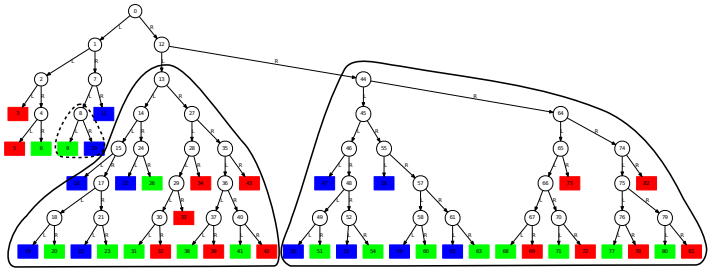


Figure 4-10: The binary tree of the sixth stage.

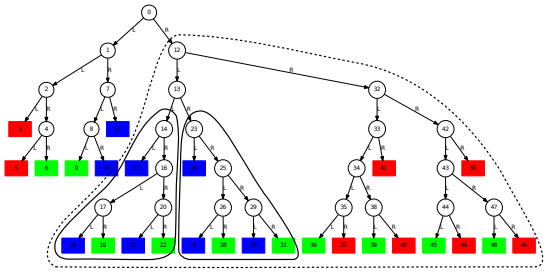


Figure 4-11: The binary tree of the seventh stage.

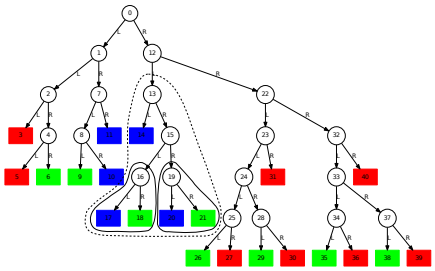


Figure 4-12: The binary tree of the eighth stage.

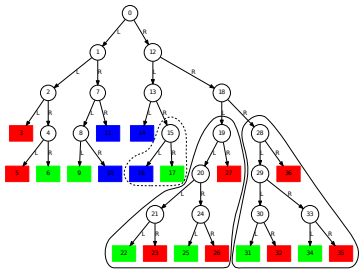


Figure 4-13: The binary tree of the ninth stage.

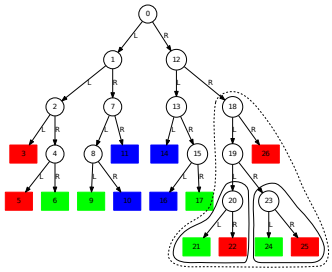


Figure 4-14: The binary tree of the tenth stage.

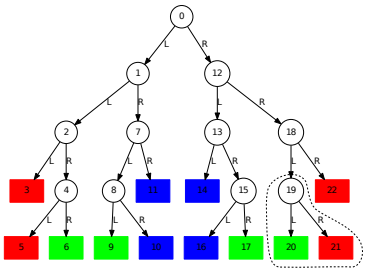


Figure 4-15: The binary tree of the eleventh stage.

4-5 Summary

In this chapter it has been shown how a recursive version of the SIVIA algorithm can be used to generate approximate set inversions represented by binary trees.

Furthermore it has been shown how set operations like unions, intersections, complement, membership tests, and orthogonal projections can be constructed that operate on sets represented by the aforementioned binary trees.

The orthogonal projection algorithm presented in this chapter has, to the author's best knowledge, not yet been described before in the literature.

Also, the set operations presented in this chapter preserve the `UNDETERMINED` labels, this in contrast with the set operations described in [16] and [22] where this information is discarded.

Chapter 5

Controller synthesis

This chapter shows how, given a difference equation describing (or approximating) the dynamics of a plant, a cost function, a parametrised controller, and optionally an inequality constraint on a function of the state and the control action, an optimal state-dependent parameter function can be constructed. This state-dependent parameter function consists of a partitioning (a regular subpaving) of the state space, where for each element of the partitioning an optimal controller parameter is specified.

Section 5-1 introduces the *controller synthesis problem* and reformulates it in such a way that it can be solved using the steps described in Section 5-2.

Section 5-3 shows how the steps described in Section 5-2 can be approximated using modified versions of the algorithms described in Chapters 3 and 4 in order to construct a practical version of the algorithm.

Section 5-4 concludes this chapter with a summary of the presented information.

5-1 The controller synthesis problem

The controller synthesis problem considered in this thesis is formulated as follows: find for each time step k the value of the state-dependent parameter function, Φ, that minimises the sum of the current and future costs, i.e. minimise:

$$V(x(0)) = \min_{\left\{\Phi\left(x(k)\right)\right\}_{k=0}^{\infty}} \sum_{k=0}^{\infty} \Big(J_{\mathrm{x}}(x(k)) + J_{\mathrm{u}}(u(k)) \Big) \tag{5-1}$$

subject to the dynamics of a system:

$$x(k+1) = f(x(k), u(k)) \tag{5-2}$$

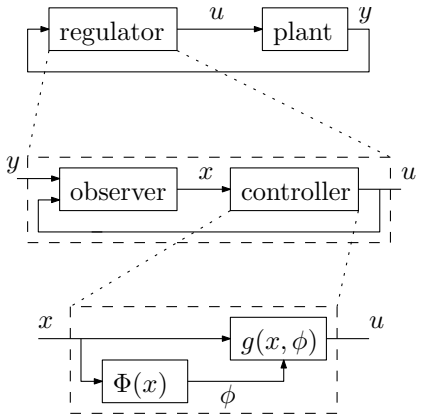


Figure 5-1: Breakdown of a regulator plant interconnect.

using a parametrised controller:

$$u(k) = g\Big(x(k), \Phi\big(x(k)\big)\Big) \tag{5-3}$$

and a inequality constraint on the state and control action:

$$0 \geq h\big(x(k), u(k)\big) \tag{5-4}$$

In order for the presented synthesis method to work, the functions f, h, g, J_{x}, and J_{u}, should be represented as *real-valued closed-form expressions*, since this is a sufficient condition to guarantee that their interval extensions can be constructed, as shown in Section 2-2. The functions J_{x} and J_{u} are assumed to be non-negative.

Figure 5-1 shows the functional block diagram of a regulator-plant interconnect and a breakdown of its sub-blocks with the bottom block containing the pre-specified parametrised controller $g(x, \phi)$ and the to be generated state-dependent controller parameters $\Phi(x)$. It is assumed that the state x is observable.

In [2] the following definition of the *principle of optimality* is given:

Theorem 1. Principle of Optimality. *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

Due to the *principle of optimality* (Theorem 1) we can rewrite (5-1) as:

$$V\big(x(0)\big) = \min_{\Phi\left(x(0)\right)} \left\{ J_{\mathrm{x}}\big(x(0)\big) + J_{\mathrm{u}}\big(u(0)\big) + \underbrace{\min_{\left\{\Phi\left(x(k)\right)\right\}_{k=1}^{\infty}} \sum_{k=1}^{\infty} \Big(J_{\mathrm{x}}\big(x(k)\big) + J_{\mathrm{u}}\big(u(k)\big)\Big)}_{V\left(x(1)\right)} \right\} \tag{5-5}$$

By substituting the minimization of the sum in (5-5) with (5-1) (with $x(1)$ instead of $x(0)$) and moving $J_x\big(x(0)\big)$ outside the minimization section, which is allowed since the location of

the minimum of $V(x(0))$ is independent of $J_x(x(0))$, we obtain:

$$V(x(0)) = J_{\mathrm{x}}(x(0)) + \min_{\Phi(x(0))} \Big\{ J_{\mathrm{u}}(u(0)) + V(x(1)) \Big\} \tag{5-6}$$

If we now rewrite (5-6) for the general case, we obtain:

$$V(x(k)) = J_{\mathrm{x}}(x(k)) + \min_{\Phi(x(k))} \Big\{ J_{\mathrm{u}}(u(k)) + V(x(k+1)) \Big\} \tag{5-7}$$

Expressions (5-5), (5-6), and (5-7) are subject to (5-2), (5-3), and (5-4).

Expression (5-7) is the *Bellman equation* with a discount factor of 1, with the part of the cost function associated with the current state moved outside the minimisation section.

5-2 Idealised solution

In this section it shown how an idealised, and simplified, version of the controller synthesis algorithm can be constructed that solves (5-7). The implementation shown here is idealised since it uses a number of operations that, for the general case, can only be approximated, namely: solving backwards reachability, and solving constrained nonlinear optimisation problems. In Section 5-3 it is shown how this idealised version of the synthesis algorithm can be approximated using interval algorithms.

In order to use (5-7) as the basis for our controller synthesis algorithm we regard its states and control actions as ranges (see Definition 1), or sets, rather than reals, i.e.:

$$V_k(X_k) = J_{\mathrm{x}}(X_k) + \min_{\Phi_k(X_k)} \{ J_{\mathrm{u}}(u(k)) + V_{k+1}(X_{k+1}) \} \tag{5-8}$$

with $X_k \subset \mathbb{R}^m$ and where $V_k(X_k)$ is the range of the value function evaluation of the set of states X_k, i.e., $V_k(X_k) = \{V_k(x) \mid x \in X_k\}$.

The subscript k for V, X and Φ is used to distinguish the costs, states, and parameters obtained at iteration $N-k$ of the synthesis algorithm, where N is the time step at which the target state(s) are reached.

First we merge f, h and J_{u} with g in order to obtain expressions in terms of the state and controller parameters:

$$f_{\mathrm{g}}(x,\phi) \stackrel{\mathrm{def}}{=} f(x, g(x,\phi)) \tag{5-9}$$

$$h_{\mathrm{g}}(x,\phi) \stackrel{\mathrm{def}}{=} h(x, g(x,\phi)) \tag{5-10}$$

$$J_{\mathrm{ug}}(x,\phi) \stackrel{\mathrm{def}}{=} J_{\mathrm{u}}(g(x,\phi)) \tag{5-11}$$

and define a target set X_{target} representing the desired state, or set of states, of the system. The value of V is by definition zero if the state is a member of this target set, i.e., $V_N(X_{\mathrm{target}}) \stackrel{\mathrm{def}}{=} 0$, where, again, N is the time step at which the target set, X_{target}, is reached. After having defined the target set we can obtain the set of states, X_{N-1}, that given the dynamics (5-2)

and constraints (5-4) can reach X_{target} by solving the *backwards reachability* problem by first solving the *set inversion problem*:

$$(X_{\text{uc}}, \Phi_{\text{uc}})_{N-1} = f_{\text{g}}^{-1}\left(X_{\text{target}}\right) \tag{5-12}$$

obtaining the joint range of the states and controller parameters, $(X_{\text{uc}}, \Phi_{\text{uc}})_{N-1}$, that can reach the target set, X_{target}, in one time step.

Next we compute the joint range of the states and controller parameters that satisfy (5-4):

$$(X_{\text{cs}}, \Phi_{\text{cs}}) = h_{\text{g}}^{-1}\big((-\infty, 0]\big) \tag{5-13}$$

and taking the intersection, of the *joint ranges* resulting from (5-12) and (5-13):

$$(X, \Phi)_{N-1} = (X_{\text{uc}}, \Phi_{\text{uc}})_{N-1} \cap (X_{\text{cs}}, \Phi_{\text{cs}}) \tag{5-14}$$

resulting in the joint range of the states and control actions, $(X, \Phi)_{N-1}$, that can both reach X_{target} in one time step, i.e. at time step N, and satisfy (5-4).

In addition to calculating $(X_{\text{cs}}, \Phi_{\text{cs}})$ to satisfy (5-4) an additional constraint can[1] be added so that the control actions will be limited to those that will not will steer the state outside the range X_{range} for which the controller is constructed:

$$(X_{\text{cs}}, \Phi_{\text{cs}}) = h_{\text{g}}^{-1}\big((-\infty, 0]\big) \cap f_{\text{g}}^{-1}(X_{\text{range}}) \tag{5-15}$$

We can now obtain the set of all states, X_{N-1}, that can reach X_{target} at time step N, from $(X, \Phi)_{N-1}$ by projecting $(X, \Phi)_{N-1}$ on the state space using the projection operator, π_{x}:

$$X_{N-1} = \pi_{\text{x}}\big((X, \Phi)_{N-1}\big) \tag{5-16}$$

Figure 5-2 shows an example of $(X_{\text{uc}}, \Phi_{\text{uc}})_{N-1}$, $(X_{\text{cs}}, \Phi_{\text{cs}})$, $(X, \Phi)_{N-1}$, and X_{N-1}, and how they relate to each other.

Having obtained X_{N-1} we can calculate the optimal controller parameters for time step $N-1$, $\Phi_{N-1}(X_{N-1})$ (Figure 5-3a) and the associated value function $V_{N-1}(X_{N-1})$ (Figure 5-3b) for X_N and the optimal controller parameters $\Phi_{N-1}(X_{N-1})$ since $V_N(X_{\text{target}})$ is zero and thus can be eliminated from (5-8) to obtain:

$$\begin{aligned} V_{N-1}(X_{N-1}) = & J_{\text{x}}\left(X_{N-1}\right) + \min_{\Phi_{N-1}\left(X_{N-1}\right)} \Big\{ J_{\text{ug}}\Big(X_{N-1}, \Phi_{N-1}(X_{N-1})\Big) \\ & \Big| \ \forall x \in X_{N-1} : (x, \Phi_{N-1}(x)) \subseteq (X, \Phi)_{N-1} \Big\} \end{aligned} \tag{5-17}$$

We assume that the associations between X, Φ, and V, are stored during the various operations mentioned above; in Section 5-3 it is shown how this is achieved.

[1] Note that this constraint is also implicitly satisfied during the calculation of $f_{\text{g}}^{-1}\big(X_k\big)$, but is likely to be beneficial when added explicitly when a version of SIVIA is used that supports pre-computed constraints on its solution set.

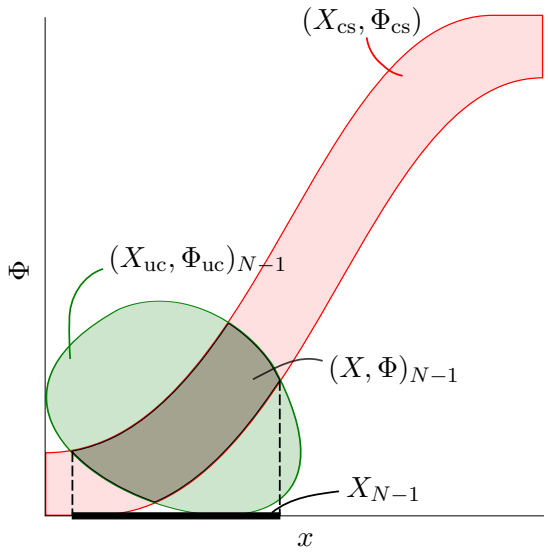
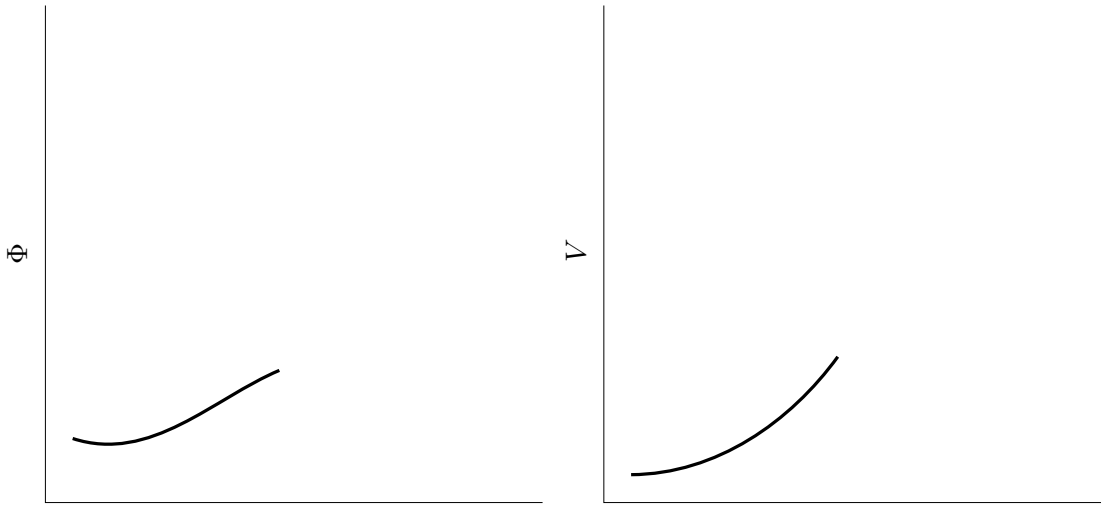


Figure 5-2: Example of the various sets present in the first iteration.



(a) X_{N-1} versus $\Phi_{N-1}(X_{N-1})$. **(b)** X_{N-1} versus $V_{N-1}(X_{N-1})$.

Figure 5-3: Optimal controller parameters and their value function for the first iteration.

We now solve the backwards reachability problem again, to find the set of states, X_{N-2}, that can reach X_{N-1} in one time step:

$$(X_{\text{uc}}, \Phi_{\text{uc}})_{N-2} = f_{\text{g}}^{-1}(X_{N-1}) \tag{5-18}$$

$$(X, \Phi)_{N-2} = (X_{\text{uc}}, \Phi_{\text{uc}})_{N-2} \cap (X_{\text{cs}}, \Phi_{\text{cs}}) \tag{5-19}$$

$$X_{N-2} = \pi_{\text{x}}((X, \Phi)_{N-2}) \tag{5-20}$$

Figure 5-4 shows an example of $(X_{\text{uc}}, \Phi_{\text{uc}})_{N-2}$, $(X_{\text{cs}}, \Phi_{\text{cs}})$, $(X, \Phi)_{N-2}$, and X_{N-2}, and how they relate to each other.

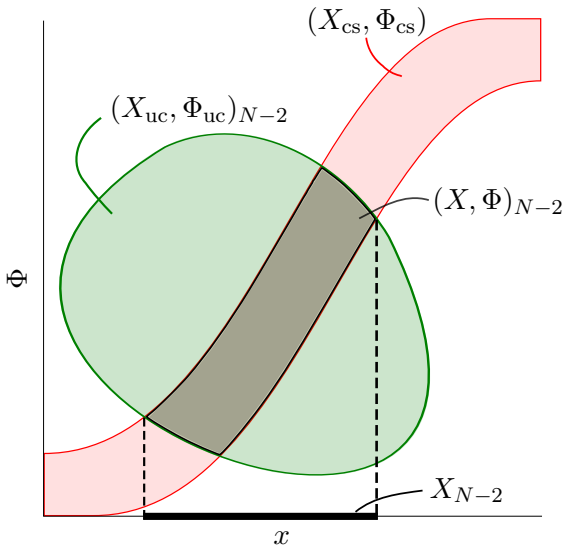


Figure 5-4: Example of the various sets present in the second iteration.

Knowing the set of states, X_{N-2}, that can reach the set of states X_{N-1} and the value function for X_{N-1}, $V_{N-1}(X_{N-1})$ we have all the required information needed to calculate the value function V for X_{N-2}:

$$V_{N-2}(X_{N-2}) = J_x(X_{N-2}) + \min_{\Phi_{N-2}(X_{N-2})} \Big\{ J_{ug}\Big(X_{N-2}, \Phi_{N-2}(X_{N-2})\Big) + V_{N-1}(X_{N-1}) \;\Big|\; \forall x \in X_{N-2} : (x, \Phi_{N-2}(x)) \subseteq (X, \Phi)_{N-2} \Big\} \tag{5-21}$$

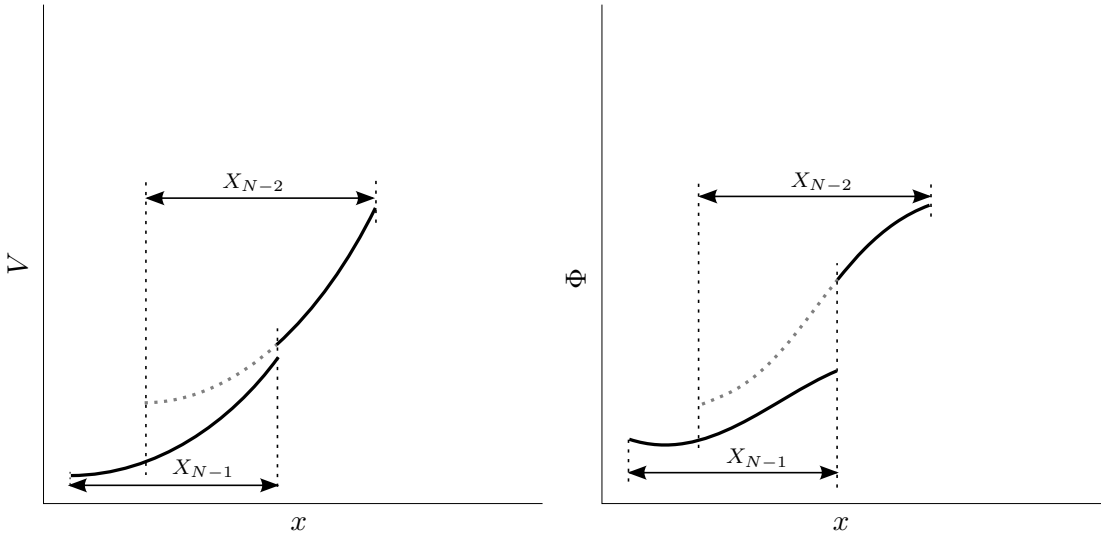
thus obtaining the optimal controller parameters, $\Phi_{N-2}(X_{N-2})$ (top curve in Figure 5-5b), associated with X_{N-2} and their accompanying value function $V_{N-2}(X_{N-2})$ (top curve in Figure 5-5a).

The next step is to merge the current and previous set of states, value functions and controller parameters, creating the set of states X^*_{N-2}, value function $V^*_{N-2}(X^*_{N-2})$ and controller parameters $\Phi^*_{N-2}(X^*_{N-2})$ for two time steps.

We use the minima of $V_{N-2}(X_{N-2})$ and $V_{N-1}(X_{N-1})$, and their associated controller parameters, for the states that X_{N-2} and X_{N-1} have in common, i.e., for states that can reach X_{target} (X_N) in either one, or two time steps, we use the control action that result in the lowest V for these states.

This is illustrated in Figure 5-5; the bottom curve in Figure 5-5b represents the prior controller parameters $\Phi_{N-1}(X_{N-1})$ and the top curve represents $\Phi_{N-2}(X_{N-2})$ versus X_{N-1} and X_{N-2}. The controller parameters in the dashed section of the curve are not used in the merged controller since their accompanying value function (the dashed curve in Figure 5-5a) does not improve upon the existing value function for these states. The (discontinuous) solid lines represent the merged value function and controller parameters.

By repeating the previous steps we obtain a new set of backwards reachable states, X_{N-3}, controller parameters, $\Phi_{N-3}(X_{N-3})$ and value function $V_{N-3}(X_{N-3})$, which are merged with



(a) Value function versus state.

(b) Optimal controller parameters versus state.

Figure 5-5: Optimal value function, $V^*_{N-2}\big(X^*_{N-2}\big)$, and its controller parameters, $\Phi^*_{N-2}\big(X^*_{N-2}\big)$, for the second iteration (time step $N-2$).

the previous results to obtain X^*_{N-3}, $\Phi^*_{N-3}\big(X^*_{N-3}\big)$, and $V^*_{N-3}\big(X^*_{N-3}\big)$:

$$X^*_{N-3} = X_{N-3} \cup X^*_{N-2} \tag{5-22}$$

$$V^*_{N-3}\big(X^*_{N-3}\big) = \text{merge}_V\Big(V_{N-3}\big(X_{N-3}\big),\ V^*_{N-2}\big(X^*_{N-2}\big)\Big) \tag{5-23}$$

$$\Phi^*_{N-3}\big(X^*_{N-3}\big) = \text{merge}_\Phi\Big(\Phi_{N-3}\big(X_{N-3}\big),\ \Phi^*_{N-2}\big(X^*_{N-2}\big)\Big) \tag{5-24}$$

Note that in the implementation of the synthesis algorithm the merge_V and merge_Φ functions are part of the function that calculates the union (see Section 5-3-8) and are not separate functions.

This process continues until the sets of controller parameters Φ^* represent the solution of (5-1), s.t. (5-2) and (5-4).

For completeness we now give the previous steps for the general case:

First we determine the joint range of the states and controller parameters that can reach the set of states of the previous time step given the dynamics of the system (5-2), while satisfying the imposed constraints (5-4):

$$(X, \Phi)_k = f_{\text{g}}^{-1}(X_{k+1}) \cap (X_{\text{cs}}, \Phi_{\text{cs}}) \tag{5-25}$$

with $(X_{\text{cs}}, \Phi_{\text{cs}})$ determined by (5-13) or (5-15).

Knowing the joint range of the states and controller parameters we obtain the range of the states by projecting the joint range onto the state space:

$$X_k = \pi_{\text{x}}\big((X, \Phi)_k\big) \tag{5-26}$$

For the obtained states we find the optimal value function and associated controller parameters (for the current time step):

$$V_k(X_k) = J_{\mathrm{x}}(X_k) + \min_{\Phi_k(X_k)} \left\{ J_{\mathrm{ug}}\Big(X_k, \Phi_k(X_k)\Big) + V^*_{k+1}(X_{k+1}) \;\middle|\; \forall x \in X_k : (x, \Phi_k(x)) \subseteq (X, \Phi)_k \right\} \tag{5-27}$$

where (5-12), (5-13), and (5-14) are merged into (5-25).

The coverage of the state space by the controller at time step k, X^*_k, is determined by the union of the sequence of sets of backwards reachable states up to X_N:

$$X^*_k = X_k \cup X^*_{k+1} \tag{5-28}$$

with $X^*_N = X_{\mathrm{target}}$.

The value function, $V^*_k(X^*_k)$, associated with X^*_k is determined by merging the value function at time step k with the value functions associated with the coverage of the state space of the subsequent time steps:

$$V^*_k(X^*_k) = \mathrm{merge}_V\Big(V_k(X_k),\ V^*_{k+1}(X^*_{k+1})\Big) \tag{5-29}$$

with $V^*_N(X^*_N) = \emptyset$.

The controller parameters, $\Phi^*_k(X^*_k)$, associated with X^*_k are determined by merging the set of controller parameters at time step k with the sets of controller parameters associated with the coverage of the state space of the subsequent time steps:

$$\Phi^*_k(X^*_k) = \mathrm{merge}_\Phi\Big(\Phi_k(X_k),\ \Phi^*_{k+1}(X^*_{k+1})\Big) \tag{5-30}$$

Approaches for finding solutions of the *Bellman equation* like the one described above, where we start at the goal state(s), X_{target}, and gradually work our way back to cover all states, are known as *backward induction* which [3] defines as:

> The process of solving multi-stage decision problems by finding the optimal choice in the final stage conditional on earlier choices, and then working back to the beginning taking one stage at a time.

5-3 Approximate solution using interval methods

In this section we describe how a practical, but approximate, implementation of the algorithm presented in Section 5-2 can be obtained using interval methods.

The process starts by specifying a target set. The aim of controller is to steer the state of the controlled process to this set of states. An inner approximation of the target set is represented using a regular subpaving, which in turn is represented by a binary tree

The controller synthesis algorithmm shown in Algorithm 14 has four main functions:

- `setInvertCost` function, located at the line marked (1a), approximates (5-25).
- `projectPavingCost` function, located at the line marked (2a), implements (5-26).
- `optimizeLists` function, located at the line marked (3a), approximates (5-27).
- `setUnionCost` function, located at the line marked (5a), implements (5-28), (5-29), and (5-30).

The remaining functions used in the synthesis algorithm are support functions for the aforementioned main functions.

5-3-1 Data structure

The binary tree described in Section 4-1 forms the basis of the data structure used in this section, but with a number elements added to its (leaf) nodes:

- `box` $\in \mathbb{IR}^{m+n}$ (or $\mathbb{IR}^m$, depending on the context[2]), the interval vector associated with the current node.
- `value` $\in$ {`INSIDE`, `OUTSIDE`, `UNDETERMINED`, `UNDETERMINED_OUTSIDE`, `UNDETERMINED_INSIDE`, `UNDETERMINED_INSIDE_OUTSIDE`}; the first three membership values are identical to those used in Section 4-1, the latter tree, together with `INSIDE`, are used to label intermediate nodes, depending on the value of their child nodes.
- `cost` $\in \mathbb{IR}$, if determined, the current value[3] of $[V]$ for the states associated with the node (if value = `INSIDE`).
- `listCost` $\in \mathbb{IR}$, the smallest interval that encloses the cost elements of all the tuples in `caList`.
- `caList`[4], a set (implemented as a list) of tuples, (ϕ, `cost`), where $\phi \in \mathbb{IR}^n$ is an interval vector containing controller parameters, such that $(x, \phi) \subseteq (X, \Phi)_k$, and `cost` $\in \mathbb{IR}$, is an interval containing the result of the evaluation of $[J_{\text{ug}}](x, \phi) + [V^*_{k+1}]\Big([f_{\text{g}}](x, \phi)\Big)$ where the interval vector $x \in \mathbb{IR}^m$ is the set of states associated with the current node.
- `left`, the left child node, containing the lower half of the bisected parent box.
- `right`, the right child node, containing the upper half of the bisected parent box.

An example of a binary tree consisting of a root node with two child nodes, using the elements mentioned above, is given in Figure 5-6.

[2] $\mathbb{IR}^{m+n}$ if the box represents (X, Φ), and $\mathbb{IR}^m$ if the box represents only X

[3] Note that is `cost` also used to (temporarily) store intermediate cost information by the `setInvertCost` function.

[4] The name `caList` was initially an abbreviation of control action list since originally the list stored control actions and their associated costs, but later on it was changed to contain controller parameters and their associated costs.

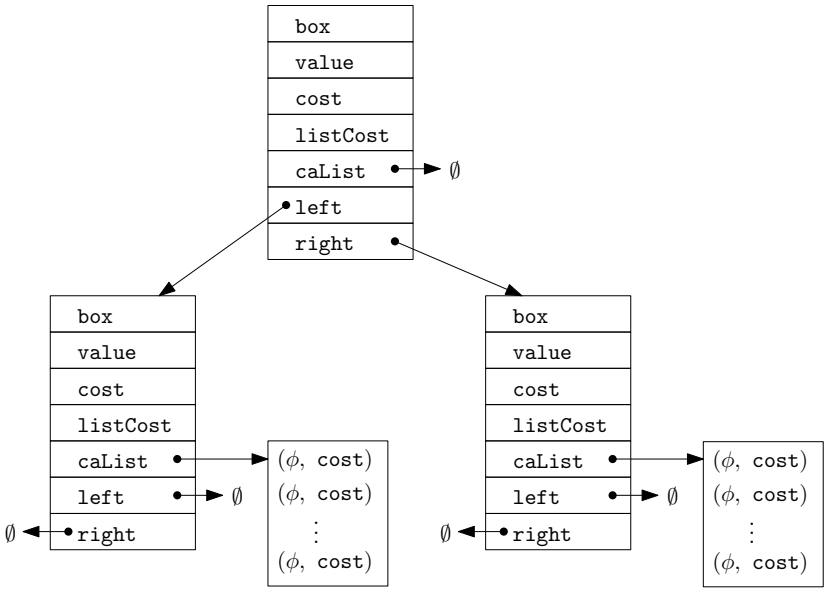


Figure 5-6: Example of a binary tree containing the described elements.

5-3-2 Initialisation

This section describes the initialisation of the practical implementation of the controller synthesis algorithm described in the previous paragraphs.

First we specify an interval vector, `box`, that encloses the ranges of the controller parameters and states for which the controller has to be constructed. States or controller parameters outside of this interval vector are not considered[5] during the controller synthesis.

Also an Regular Sub-Paving (RSP) of π_{x}(`box`), `targetpaving`, that approximates the target states, X_{target}, is constructed. This can for example be implemented by specifying an inequality function h_{target} such that:

$$\forall x \in X_{\mathrm{target}} : h_{\mathrm{target}}(x) \leq 0 \tag{5-31}$$

which can be rewritten as a set-inversion problem (in a similar way as (5-13)):

$$X_{\mathrm{target}} = h_{\mathrm{target}}^{-1}((-\infty, 0]) \tag{5-32}$$

which can be approximated by an RSP generated using the SIVIA algorithm specified in Section 4-1.

At this point the RSP representing the target set is identical to the RSPs used in Chapter 4, i.e., a binary tree with leaf nodes labelled `INSIDE`, `OUTISDE`, or, `UNDETERMINED`. In order to use the RSP in the controller synthesis algorithm we add a number of entries and labels to the nodes of the binary tree representing the RSP. The first of these new entries is the cost entry `cost`.

[5]However it is possible to 'virtually' extend the range of the states for which the controller is constructed for some systems as will be shown in Sub-section 6-1.

In the RSPs representing the various sets of states, $X_{\text{target}}, X_0, \ldots, X_k, X_0^*, \ldots, X_k^*$, the `cost` entry is used to store the value to $[V]$ associated with the states that correspond to the node if the node is labelled `INSIDE`, or `NaN` otherwise, where `NaN` is used to indicate that (currently) no value of $[V]$ exists or that value is not available for the states associated with the node.

In addition to specifying X_{target} interval extensions of f_{g}, J_{x}, J_{ug}, and optionally h_{g} are constructed by, for example, using one of the approaches described in Section 2-2.

The RSP representing the target set is used as an input argument the controller synthesis algorithm described in Section 5-3-3.

5-3-3 Controller synthesis algorithm

This subsection gives an overview of the functions used in the controller synthesis algorithm shown in Algorithm 14 for each function a short overview of its purpose is given and a reference to the algorithm that implements the function.

At the beginning of the overview of each function the relevant location of the function is specified, e.g., when describing Algorithm 14 (0a) refers to the line marked (0a) in Algorithm 14.

(0a): The controller synthesis algorithm starts by setting the cost entries of the nodes labelled `INSIDE` of the target set X_{target} to zero, the `addZeroCost` function implements this.

(0b): In the next step of the synthesis algorithm the labels of the leaf nodes of the binary tree representing the RSP containing the target set are propagated back to the root node of the tree using the `backPropLabels` function described in Algorithm 15 for later use with the `projectPavingCost` function.

(0c),(0d): Both the set of backwards reachable states, X_N, and the coverage of the controller and associated controller parameters, X_N^* and Φ_N^* respectively, are initialised with the value of the target paving before entering the main loop of the synthesis algorithm.

(0e): For initialisation purposes the `initialize` flag is set to `TRUE` before entering the main loop; the `initialize` flag is used to distinguish the first iteration of the loop from following iterations.

(0f): At the start of the main loop of the synthesis algorithm we call the `parameterScheduler` function; this function returns, for each value i of the loop counter the `maxDepth` and `maxRefineDepth` parameters that are used in the `setInvertCost` and `optimizeLists` functions to respectively specify the recursion depth of the set inversion and the maximum number of bisections of the parameter interval vector.

(1a): Next we call the `setInvertCost` function described in Algorithm 18. This function implements an approximation of (5-25). At this point we restrict ourselves to the case when $h_{\text{g}}^{-1} \in \mathbb{IR}^{n+m}$, i.e., when the joint range of the states and controller parameters (see (5-13)), $(X_{\text{cs}}, \Phi_{\text{cs}})$, is an interval vector. This restriction allows us to avoid having to explicitly evaluate (5-14) by using $(X_{\text{cs}}, \Phi_{\text{cs}})$ as the root box of the RSP that holds the approximation of $(X, \Phi)_k$ in the set inversion algorithm, i.e., the set inversion algorithm does not consider values outside $(X_{\text{cs}}, \Phi_{\text{cs}})$.

Note that besides determining the approximation of $(X, \Phi)_k$ the `setInvertCost` function also uses the `cost` entry of the nodes labelled `INSIDE` to temporarily store the value of

$[J_{\text{ug}}](x,\phi)+[V^*_{k+1}]([f_{\text{g}}](x,\phi))$, where[6] $x = \pi_{\text{x}}(\texttt{node.box})$ and $\phi = \pi_{\Phi}(\texttt{node.box})$ and `node.box` is the joint range of the state and parameter vectors associated with the current node. The rationale for doing this is that evaluation of $[V^*_{k+1}]([f_{\text{g}}](x,\phi))$ is done implicitly during the membership test of $[f_{\text{g}}](x,\phi)$ in the set inversion algorithm. Since its value is also used later on, duplicate evaluations of $[J_{\text{ug}}](x,\phi) + [V^*_{k+1}]([f_{\text{g}}](x,\phi))$ can thus be prevented by storing its value in `node.cost`.

The 0 in the argument of `setInvertCost` is used to initialize the recursion depth counter.

(1b): The `addBoxValues` function shown in Algorithm 22 sets the (correct) values of the `box` entries for the nodes in the binary tree that represents the RSP containing the approximation of X_k.

The 0 in the argument of `addBoxValues` is used to initialize the recursion depth counter.

(1c): The `addListValues` function shown in Algorithm 23 initialises the `caList` entry of the nodes labelled `INSIDE` with the appropriate value of the control action and its associated cost, $(\pi_{\Phi}(\texttt{node.box}), \texttt{node.cost})$.

The 2 in the argument of `addListValues` is used to specify the index of `node.box` values associated with the controller parameter.

(1d): In preparation to the evaluation of the set projection function, `projectPavingCost`, the labels of the leaf nodes are propagated back to the root node using the `backPropLabels` function described in Algorithm 15.

(2a): By projecting the approximation of $(X,\Phi)_k$ on the state space using the `projectPavingCost` function defined in Algorithm 24 we obtain approximations of X_k and $\Phi_k(X_k)$. The `projectPavingCost` function also merges the parameter lists of the leaf nodes labelled `INSIDE` containing all the entries associated with the states associated with the respective leaf node.

The 2 in the argument of `projectPavingCost` is used to specify along which axis the projection has to take place; in this case along the axis associated with the controller parameter.

(2b): After the projection we again propagate the labels of the leaf nodes of the binary tree representing the RSP representing X_k back to the root node of the binary tree for later use.

(2c): Before executing the main optimisation function, `optimizeLists`, using the `cleanupLists` function described in Algorithm 17, we remove all `caList` entries of which the cost is strictly suboptimal, i.e., if $\texttt{cost}^*$ is defined as the lowest value of the upper bounds of the `cost` entries of the elements in `caList`, all entries for which the lower bound of the `cost` element exceeds $\texttt{cost}^*$ are removed from `caList`.

(3a): The `optimizeLists` function described in Algorithm 26 further bisects the states (it creates child nodes in the binary tree) and evaluates, updates, and optionally removes `caList` entries from the nodes associated with the bisected states using the `optimizeList` function described in Algorithm 28.

(3b): The `decimateLists` function described in Algorithm 29 removes all entries from each of the `caList` entries in the binary tree representing the RSP that contains the approximation

[6] π is the projection operator, with the subscript specifying the subspace onto which has to be projected. Since in this case `node.box` is an interval vector this is equivalent to selecting the appropriate index of the `node.box` interval vector.

of X_k except for the entry with the lowest upper bound. Note that this function can be omitted if the `decimateList` function is used in Algorithm 26, and if Algorithm 26 is used, i.e., `decimateLists` only has to be used when no explicit optimisation is used.

(3c): The `calculateCosts` function described in Algorithm 30 calculates $[V_k](X_k)$ by evaluating $[J_x]$(`node.box`) + `node.listCost` and storing the result in `node.cost` for each node of the binary tree representing the RSP that contains the approximation of X_k labelled `INSIDE`.

(3d): The `backPropCost` function described in Algorithm 31 propagates, to the extent possible, the `node.cost` values of the nodes labelled `INSIDE` back the the root node for later use.

Depending on the value of the `initialize` flag we distinguish between two cases:

- If the flag is set to `TRUE`: (4a),(4b): the binary tree representing the RSP that contains the approximation of X_k^*, $[V_k^*](X_k^*)$, and $\Phi_k^*(X_k^*)$ is initialised with a copy of the binary tree representing the RSP that contains the approximation of X_k, $[V_k](X_k)$, and $\Phi_k(X_k)$; next the `initialize` flag is set to `FALSE`.
- Otherwise, if the `initialize` flag is set to `FALSE`: (5a),(5b),(5c),(5d),(5e): the binary tree representing the RSP that contains the approximation of X_k^*, $[V_k^*](X_k^*)$, and $\Phi_k^*(X_k^*)$ is initialized by merging X_{k+1}^*, $[V_{k+1}^*](X_{k+1}^*)$, and $\Phi_{k+1}^*(X_{k+1}^*)$ with the binary tree representing the RSP that contains the approximation of X_k, $[V_k](X_k)$, and $\Phi_k(X_k)$; moreover, the `decimateLists`, `calculateCosts`, `backPropCost`, and `backPropLabels` functions are evaluated for the resulting binary tree.

(6a),(6b): At the end the iteration the results of the current iteration are re-assigned to be the input of the next iteration.

(7): When the maximum number of iterations has been reached, the binary tree representing the controller paving is returned.

Algorithm 15 propagates the labels of the leaf nodes of the binary tree given as an argument back to the root node of the tree using the rules given in Table 5-1 by assigning to each intermediate node a value that depends on the value of its child nodes, with `lValue` and `rValue` being respectively the value of the left and right child nodes. In Table 5-1 O, U, I, UO, UI and UIO are used as shorthand for respectively the node values `OUTSIDE`, `UNDETERMINED`, `INSIDE`, `UNDETERMINED_OUTSIDE`, `UNDETERMINED_INSIDE` and `UNDETERMINED_INSIDE_OUTSIDE`. The two X's in Table 5-1 are used to indicate a forbidden combination of values for `lValue` and `rValue`. Since the boxes associated with `lValue` and `rValue` share an edge, one of their values cannot be `OUTSIDE` while the other is `INSIDE` (and vice versa) since this would imply that their shared edge simultaneously can have the value `OUTSIDE` and `INSIDE`. An example of an RSP and its representation by a binary tree where the leaf labels are backpropagated is given in Figures 5-7 and 5-8.

The values of the intermediate nodes are used in the `setUnionCost` function (shown in Algorithm 32) to determine whether a sub-tree has to be traversed or not (see the description of Algorithm 32 for more details).

Algorithm 16 is used to create a copy of the tree specified in its argument.

Algorithm 17 describes the `cleanupLists` function. The `cleanupList` function removes all entries, from the argument list, for which the lower bound of the cost is larger than the lowest

Algorithm 14: controllerSynth(targetPaving, box, $[f_g]$, $[J_x]$, $[J_{ug}]$, iterations, parameterScheduler)

input : An RSP of X_{target}, targetPaving, an interval vector specifying the ranges of Φ and X, box, an interval extension of f_g, $[f_g]$, an interval extension of J_x, $[J_x]$, an interval extension of J_{ug}, $[J_{ug}]$, the maximum number of iterations, iterations, and parameter scheduling function, parameterScheduler

output : A regular subpaving of box representing X_0^* containing $V_0^*(X_0^*)$ and $\Phi_0^*(X_0^*)$.

```
begin
(0a)  addZeroCost(targetPaving)
(0b)  backPropLabels(targetPaving)
(0c)  setProjection ⟵ copyTreeCost(targetPaving)
(0d)  controller ⟵ copyTreeCost(targetPaving)
(0e)  initialize ⟵ TRUE
      for i = 1 to iterations do
(0f)      (maxDepth, maxRefineDepth) ⟵ parameterScheduler(i)
(1a)      tempSet ⟵ setInvertCost(box, [f_g], [J_ug], 0, maxDepth, setProjection)
(1b)      addBoxValues(tempSet, box, 0)
(1c)      addListValues(tempSet, 2)
(1d)      backPropLabels(tempSet)
(2a)      nextSetProjection ⟵ projectPavingCost(tempSet, 2)
(2b)      backPropLabels(nextSetProjection)
(2c)      cleanupLists(nextSetProjection)
(3a)      optimizeLists(nextSetProjection, [J_ug], [f_g], controller, controller, maxRefineDepth)
(3b)      decimateLists(nextSetProjection)
(3c)      calculateCosts(nextSetProjection, controller, [J_x])
(3d)      backPropCost(nextSetProjection)
          if initialize = TRUE then
(4a)          nextController ⟵ copyTreeCost(nextSetProjection)
(4b)          initialize ⟵ FALSE
          else
(5a)          nextController ⟵ setUnionCost(controller, nextSetProjection)
(5b)          decimateLists(nextController)
(5c)          calculateCosts(nextSetProjection, controller, [J_x])
(5d)          backPropCost(nextController)
(5e)          backPropLabels(nextController)
(6a)      setProjection ⟵ nextSetProjection
(6b)      controller ⟵ nextController
(7)   return controller
```

Table 5-1: Backpropagation rules for node labels.

lValue \ rValue	O	U	I	UO	UI	UIO
O	O	UO	X	UO	UIO	UIO
U	UO	U	UI	UO	UIO	UIO
I	X	UI	I	UIO	UI	UIO
UO	UO	UO	UIO	UO	UIO	UIO
UI	UIO	UIO	UI	UIO	UI	UIO
UIO	UIO	UIO	UIO	UIO	UIO	UIO

Algorithm 15: The `backPropLabels(node)` algorithm.

input : The root node, `node`, of a binary tree representing a regular subpaving.
output : The binary tree with the `value` elements of each node set to the appropriate values.

begin
  if *isLeaf*(*node*) = *TRUE* **then**
    return `node.value`
  `lValue` ⟵ `backPropLabels(node.left)`
  `rValue` ⟵ `backPropLabels(node.reft)`
  if *lValue* = *rValue* **then**
    `node.value` ⟵ `lValue`
    return `node.value`
  `node.value` ⟵ `UNDETERMINED_INSIDE_OUTSIDE`
  if *lValue* = *UNDETERMINED_INSIDE_OUTSIDE* *or*
  rValue = *UNDETERMINED_INSIDE_OUTSIDE* **then**
    return `node.value`
  if *lValue* = *UNDETERMINED* **then**
    if *rValue* = *INSIDE* *or* *rValue* = *UNDETERMINED_INSIDE* **then**
      `node.value` ⟵ `UNDETERMINED_INSIDE`
    else if *rValue* = *OUTSIDE* *or* *rValue* = *UNDETERMINED_OUTSIDE* **then**
      `node.value` ⟵ `UNDETERMINED_OUTSIDE`
  else if *rValue* = *UNDETERMINED* **then**
    if *lValue* = *INSIDE* *or* *lValue* = *UNDETERMINED_INSIDE* **then**
      `node.value` ⟵ `UNDETERMINED_INSIDE`
    else if *lValue* = *OUTSIDE* *or* *lValue* = *UNDETERMINED_OUTSIDE* **then**
      `node.value` ⟵ `UNDETERMINED_OUTSIDE`
  else if *(lValue* = *OUTSIDE* *and* *rValue* = *OUTSIDE_UNDETERMINED)* *or*
  (lValue = *OUTSIDE_UNDETERMINED* *and* *rValue* = *OUTSIDE)* **then**
    `node.value` ⟵ `UNDETERMINED_OUTSIDE`
  else if *(lValue* = *INSIDE* *and* *rValue* = *INSIDE_UNDETERMINED)* *or*
  (lValue = *INSIDE_UNDETERMINED* *and* *rValue* = *INSIDE)* **then**
    `node.value` ⟵ `UNDETERMINED_OUTSIDE`
  return `node.value`

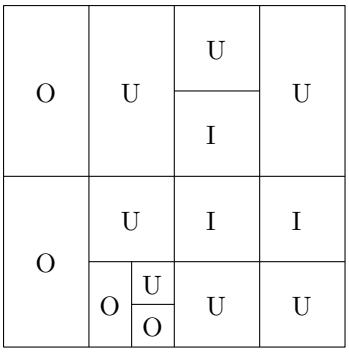


Figure 5-7: An example paving.

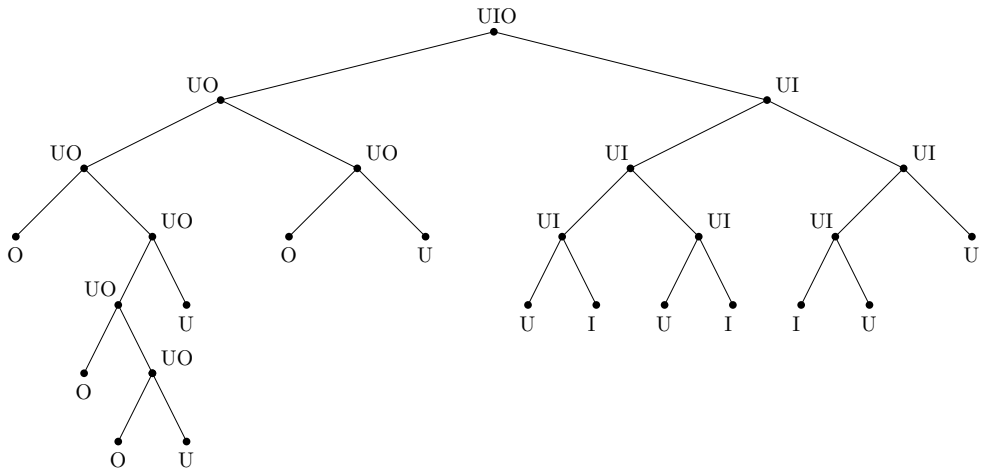


Figure 5-8: An example of a binary tree (after backpropagation of the leaf node labels) with the values of the `value` elements of each node shown.

Algorithm 16: The `copyTreeCost(nodeIn)` algorithm.

input : The root node, `nodeIn`, of a regular subpaving.
output : The root node of a copy of the argument tree.

```
begin
    nodeOut ⟵ copyNodeData(nodeIn)
    if isLeaf(node) = FALSE then
        nodeOut.left ⟵ copyTreeCost(nodeIn.left)
        nodeOut.right ⟵ copyTreeCost(nodeIn.right)
    return nodeOut
```

upper bound of any of the entries in the list. The `getListCostBounds` function returns an interval consisting of the lowest lower bound and the highest upper bound of all the cost elements of the entries in the list given as argument of the function. The `listSize` function returns the number of entries in the list given as an argument.

Algorithm 17: The `cleanupLists(node)` algorithm.

input : The root node, `node`, of a binary tree representing a regular subpaving.
output : The binary tree where the suboptimal entries of the `caList` elements have been removed.

```
begin
    if isLeaf(node) = TRUE then
        if node.value = INSIDE then
            if listSize(node.caList) > 0 then
                cleanupList(node.caList)
                node.listCost ⟵ getListCostBounds(node.caList)
    else
        cleanupLists(node.left)
        cleanupLists(node.right)
```

5-3-4 Set inversion

The set inversion algorithm described in Algorithm 18 is an extended version of Algorithm 6. The main difference is that during the membership test by `pavingMembershipAndCost` not only the membership of X_{next} with respect to Φ is returned but also the the value of $[V_{k+1}](X_{\text{next}})$. At this point the value of $[V_{k+1}](X_{\text{next}})$ is not used, but it can be used during the minimization step. The value of $[V_{k+1}](X_{\text{next}})$ is stored[7] in the `cost` element of the current node for later use.

In Algorithm 19 the pseudo-code of a set inversion algorithm that directly calculates the result of (5-25) is given. It is largely similar to (18) but also takes as an argument a paving of[8] (as determined by (5-13)). The advantage of using this approach is that only boxes inside $(X_{\text{cs}}, \Phi_{\text{cs}})$ are evaluated during the set inversion, eliminating the evaluation of boxes that are discarded anyway.

5-3-5 Membership test *and* value function

Algorithm 20 implements both the value function $[V] : (x, \Psi) \to [V](x, \Psi)$ with $x \in \mathbb{IR}^n$, $X \in \mathbb{IR}^n$, $x \subseteq X$, and $\Psi \in \mathcal{RSP}(X)$, and a *membership test* of x on the set represented by Ψ. Algorithm 20 returns the membership of x with respect to the set represented by Ψ and an interval that contains the highest upper bound and lowest lower bound of the cost of all boxes in Ψ labelled `INSIDE` that intersect with X, when the membership value equals `INSIDE`, i.e., all intersecting boxes are labelled `INSIDE`. The value `NaN` is used as a dummy value in case the result is `UNDETERMINED` or `OUTSIDE`.

Algorithm 20 shows a possible implementation of $[V]$; It is implemented by extending Algorithm 9 with code that keeps track of the upper and lower bonds of the intersecting boxes.

[7] The `cost` element of the node is used as temporary storage for this purpose; later on it is used to store the value of [V] associated with the given node.

[8] For brevity the pseudo-code uses Ξ to denote $(X_{\text{cs}}, \Phi_{\text{cs}})$.

Algorithm 18: The `setInvertCost`(`box`, $[f_g]$, $[J_{ug}]$, Ψ, `currDepth`, `maxDepth`) function.

input : An interval vector, `box` containing the joint range of the state and controller parameters to be investigated, an interval extension of the dynamics, $[f_g]$, an interval extension of the control action cost function, $[J_{ug}]$, the maximum recursion depth, `maxDepth`, the current recursion depth, `currDepth`, and a paving of the target set, Ψ.

output : A regular paving containing the approximation of $f_g^{-1}(\Psi)$.

begin

```
node ⟵ newnode()
X ⟵ π_x(box)
Φ ⟵ π_Φ(box)
X_next ⟵ [f_g](X, Φ)
(node.value, node.cost) ⟵ pavingMembershipAndCost(Ψ, X_next, 0)
if node.value = INSIDE then
    node.cost ⟵ node.cost + [J_ug](X, Φ)
    return node
else if node.value = OUTSIDE then
    return node
else
    if currDepth = maxDepth then
        node.value ⟵ UNDETERMINED
        return node
    else
        index ⟵ (currDepth mod (m + n)) + 1
        (box_L, box_R) ⟵ bisect(box, index)
        node.left ⟵ setInvertCost(box_L, [f_g], [J_ug], Ψ, currDepth + 1, maxDepth)
        node.right ⟵ setInvertCost(box_R, [f_g], [J_ug], Ψ, currDepth + 1, maxDepth)
        return reunite2(node)
```

Where $\sqcup$ is the *interval hull* operator (as defined in (2-34)). The `reunite2` algorithm shown in Algorithm 21 is based on Algorithm 5, but differs in that it does not merge nodes labelled `INSIDE` since doing so would lead to a loss of information[9].

5-3-6 Set projection

The set projection algorithm described in this section consists of two parts, the first part, shown in Algorithm 24 is a wrapper function for the second part, shown in Algorithm 25, which is the recursive section of the algorithm.

The wrapper function calls the recursive section and adjusts the values of the `box` entries, using the `addBoxValues` function, to reflect that the binary tree now represents a projection of the original tree. The `removeAxis` function is used to create the new value of the root box

[9] Since the remaining node data of the sibling nodes are likely to have different values.

Algorithm 19: The `setInvertCostConstraint`(box, $[f_g]$, $[J_{ug}]$, Ψ, Ξ, `currDepth`, `maxDepth`) function.

input : An interval vector, box containing the joint range of the state and controller parameters to be investigated, an interval extension of the dynamics, $[f_g]$, an interval extension of the control action cost function, $[J_{ug}]$, the maximum recursion depth, `maxDepth`, the current recursion depth, `currDepth`, a paving of the target set, Ψ and a paving of the constraint set Ξ.

output : A regular paving containing the approximation of $f_g^{-1}(\Psi) \cap \Xi$.

begin

```
node ⟵ newnode()
if Ξ.value = UNDETERMINED_OUTSIDE then
    newNode.value ⟵ UNDETERMINED
    newNode.cost ⟵ NaN
    return newNode
if Ξ.value = UNDETERMINED then
    newNode.value ⟵ UNDETERMINED
    newNode.cost ⟵ NaN
    return newNode
if Ξ.value = OUTSIDE then
    newNode.value ⟵ OUTSIDE
    newNode.cost ⟵ NaN
    return newNode
X ⟵ π_x(box)
Φ ⟵ π_Φ(box)
X_next ⟵ [f_g](X, Φ)
(node.value, node.cost) ⟵ pavingMembershipAndCost(Ψ, X_next, 0)
if node.value = INSIDE then
    node.cost ⟵ node.cost + [J_ug](X, Φ)
    return node
else if node.value = OUTSIDE then
    return node
else
    if currDepth = maxDepth then
        node.value ⟵ UNDETERMINED
        return node
    else
        index ⟵ (currDepth mod (m + n)) + 1
        (box_L, box_R) ⟵ bisect(box, index)
        (Ξ_L, Ξ_R) ⟵ returnChildNodes(Ξ)
        node.left ⟵ setInvertCostConstraint(box_L, [f_g], [J_ug], Ψ, Ξ_L, currDepth + 1, maxDepth)
        node.right ⟵ setInvertCostConstraint(box_R, [f_g], [J_ug], Ψ, Ξ_R, currDepth + 1, maxDepth)
        return reunite2(node)
```

Algorithm 20: The `pavingMembershipAndCost(node, X, depth)` function.

input : The root node of a regular subpaving, `node`, a box, $X \in \mathbb{IR}^n$, and the current recursion depth, `depth`.

output : The membership of X with respect to the set approximated by the paving with root node `node`, and the lowest lower bound and the highest upper bound, of all nodes labelled `INSIDE` intersecting with X.

begin

```
if isLeaf(node) = TRUE then
    if node.value = INSIDE then
        return (INSIDE, node.cost)
    else
        return (node.value, NaN)

index ⟵ (depth mod n) + 1

midpoint ⟵ node.left.box with overline, index

if X̄_index ≤ midpoint then
    return pavingMembershipAndCost(node.left, X, depth + 1)

if X̲_index ≥ midpoint then
    return pavingMembershipAndCost(node.right, X, depth + 1)

(leftValue, lCost) ⟵ pavingMembershipAndCost(node.left, X, depth + 1)

if leftValue = UNDETERMINED then
    return (UNDETERMINED, NaN)

(rightValue, rCost) ⟵ pavingMembershipAndCost(node.right, X, depth + 1)

if rightValue = UNDETERMINED then
    return (UNDETERMINED, NaN)

if leftValue = INSIDE and rightValue = INSIDE then
    return (INSIDE, lCost ⊔ rCost)

if leftValue = OUTSIDE and rightValue = OUTSIDE then
    return (OUTSIDE, NaN)
```

of the output paving by removing the axis specified by `axis` from the root box of the input paving.

The recursive section of the projection algorithm is basically Algorithm 13 with a few additions for handling the additional node entries.

5-3-7 Optimisation

The practical implementation of the minimization step of the controller synthesis algorithm (5-27), is located on the lines marked (3a) to (3d) of Algorithm 14. The minimization step consists of two main parts: the `optimizeLists` and the `optimizeList`, with the latter function being called by the prior.

Algorithm 21: The `reunite2`(node) function.

input : A binary tree node, `node`, representing a regular subpaving.
output : A binary tree node, representing a *semi-minimal* regular subpaving.

begin
  if *isLeaf(node==FALSE)* **then**
    if *isLeaf*(*node.left*) *and* *isLeaf*(*node.right*) *and* *node.left.value* = *node.right.value* **then**
      if *node.left.value* ≠ *INSIDE* **then**
        `node.value` ⟵ `node.left.value`
        `freenode(node.left)`
        `node.left` ⟵ ∅
        `freenode(node.right)`
        `node.right` ⟵ ∅
  return `node`

Algorithm 22: The `addBoxValues`(`node`, `box`, `depth`) algorithm.

input : A node, `node`, of a binary tree that represents a regular subpaving, the value of the box associated with the root node of the tree, `box` ∈ $\mathbb{IR}^n$, and the current recursion depth, `depth`.
output : The tree with `box` elements of the nodes set to the appropriate values.

begin
  `node.box` ⟵ `box`
  if *isLeaf*(*node*) = *FALSE* **then**
    `index` ⟵ (`depth` mod n) + 1
    (`lBox`, `rBox`) ⟵ `bisect(box, index)`
    `addBoxValues(node.left, lBox, depth` + 1)
    `addBoxValues(node.right, RBox, depth` + 1)

Algorithm 23: The `addListValues`(`node`, `index`) algorithm.

input : A node, `node`, of a binary tree that represents a regular subpaving, the index, `index`, of the (root) box associated with the controller parameter(s).
output : The binary tree with the `caList` and `listCost` values of its nodes set to their appropriate values.

begin
  if *isLeaf*(*node*) = *TRUE* **then**
    if *node.value* = *INSIDE* **then**
      `node.caList` ⟵ $\left\{(\pi_{\mathrm{x}}(\texttt{node.box}), \texttt{node.cost})\right\}$
      `node.listCost` ⟵ `node.cost`
    else
      `addListValues(node.left, index)`
      `addListValues(node.right, index)`

Algorithm 24: The `projectPavingCost`(Ψ, `axis`) algorithm.

input : A regular subpaving, $\Psi \in \mathcal{RSP}(X)$, with $X \in \mathbb{IR}^n$, and the axis, `axis` $\in \{1 \cdots n\}$, along which the projection has to take place.

output : Same as Algorithm 25, but with the `box` values of each node set to the appropriate value.

```
begin
    pavingOut ⟵ projectPavingCostRec(Ψ, axis, n)
    boxOut ⟵ removeAxis(Ψ.box, axis)
    addBoxValues(pavingOut, boxOut)
    return pavingOut
```

Algorithm 25: Recursive part of the `projectPavingCost` algorithm; `projectPavingCostRec`(Ψ, `axis`, `boxDim`).

input : The root node of a binary tree representing a regular subpaving, $\Psi \in \mathcal{RSP}(X)$, with $X \in \mathbb{IR}^n$, the axis, `axis` $\in \{1 \cdots n\}$, along which the projection has to take place, and the dimension of X, `boxDim`.

output : The root node of a binary tree representing a regular subpaving, $\Psi' \in \mathcal{RSP}(X')$, with $X' \in \mathbb{IR}^{n-1}$ representing the projection of the regular subpaving Ψ along the specified axis, with the `caList` elements containing the merged entries.

```
begin
    if isLeaf(nodeIn)= TRUE then
        node ⟵ newnode()
        node.value ⟵ nodeIn.value
        node.cost ⟵ nodeIn.cost
        if nodeIn.value = INSIDE then
            node.caList ⟵ copyCaList(nodeIn.caList)
        return node
    index ⟵ (currDepth mod boxDim) + 1
    if axis = index then
        node ⟵ setUnionCost(nodeIn.left, nodeIn.right)
        return projectPavingRec(node, axis, currDepth + 1, boxDim)
    else
        node ⟵ newnode()
        node.left ⟵ projectPavingRec(nodeIn.left, axis, currDepth + 1, boxDim)
        node.right ⟵ projectPavingRec(nodeIn.right, axis, currDepth + 1, boxDim)
        return reunite2(node)
```

The `optimizeLists` function described in Algorithm 26 traverses the binary tree representing the RSP that approximates the current set of backwards reachable states. For leaf nodes labelled `INSIDE` `optimizeLists` evaluates the `optimizeList` and `cleanupList` functions[10] for the `caList` element of each leaf node. If the maximum recursion depth, `maxDepth`, has

[10] The `cleanupList` function is also used in Algorithm 17.

not been reached, the state associated with the current node, `box`, is bisected to obtain `lBox` and `rBox`. Next two child nodes are created for the current node with the values of their `box` elements set to `lBox` and `rBox`. The `caList` element of the left child node is initialised with a copy of the `caList` of its parent node, the `caList` element of the right child node is initialized by reassigning the `caList` of its parent node to it[11]. After bisection the node is again evaluated by the `optimizeLists` function. When the maximum recursion depth is reached the `decimateList` function is called, this removes all entries from the list of control actions, except the one with the lowest upper bound of its cost entry.

Algorithm 26: The `optimizeLists`(`node`, $[J_{\text{ug}}]$, $[f_{\text{g}}]$, `targetPaving`, `controllerNode`, `maxDepth`, `depth`) algorithm.

input : The root node, `node`, of the RSP representing the current backwards reachable set of states, an interval extension of J_{ug}, $[J_{\text{ug}}]$, an interval extension of f_{g}, $[f_{\text{g}}]$, the RSP containing X^*_{k+1}, `targetPaving`, the node of the binary representing the RSP of X^*_{k+1} that corresponds to the the current `node`, `controllerNode`, the maximum recursion depth, `maxDepth`, and the current recursion depth, `depth`.

output : The argument RSP containing the (potentially) optimal values of $\Phi_k(X_k)$ and their associated costs.

```
begin
    if isLeaf(node) = TRUE then
        if listSize(node.caList) > 0 then
            optimizeList(node.caList, node.box, [J_ug], [f_g], targetPaving, controllerNode)
            cleanupList(node.caList)
        if depth < maxDepth then
            node.left ⟵ newnode()
            node.right ⟵ newnode()
            index ⟵ (depth mod m) + 1
            (lBox, rBox) ⟵ bisect(node.box)
            node.left.box ⟵ lBox
            node.right.box ⟵ rBox
            node.left.caList ⟵ copyCaList(node.caList)
            node.right.caList ⟵ node.caList
            node.caList ⟵ ∅
            optimizeLists(node, [J_ug], [f_g], targetPaving, cPavingRight, maxDepth, depth)
        else
            decimateList(node.caList)
    else
        (controllerNodeLeft, controllerNodeRight) ⟵ returnChildNodes(controllerNode)
        optimizeLists(node.left, [J_ug], [f_g], targetPaving, controllerNodeLeft, maxDepth, depth + 1)
        optimizeLists(node.right, [J_ug], [f_g], targetPaving, controllerNodeRight, maxDepth, depth + 1)
```

[11] In the 'real' implementation `caList` is a pointer to a double linked list. The initialization simply consists of setting the pointer `caList` pointer of the right child node to the value of the `caList` pointer of its parent node and setting the parent node pointer to `NULL`.

Algorithm 27 is used to 'virtually' extend the recursion depth of binary trees representing regular subpavings. This approach is possible due to the fact that the properties of a box in a paving extend to its sub boxes, so for example the cost (interval) associated with a box is also valid for each of its sub boxes.

Algorithm 27: The `returnChildNodes(node)` algorithm.

input : A node, `node`, of a binary tree.
output : The left and right child nodes of `node` or `node` itself if is a leaf node.

```
begin
    if isLeaf(node) = FALSE then
        return (node.left, node.right)
    else
        return (node, node)
```

The `optimizeList` function described in Algorithm 28 is called by the `optimizeLists` function. It is functionally equivalent to a single iteration of the Moore-Skelboe algorithm described in Section 3-1.

Algorithm 28: The `optimizeList`(`list`, `box`, $[J_{\text{ug}}]$, $[f_{\text{g}}]$, `targetPaving`, `controllerNode`) algorithm.

input : A list of (controller parameter, cost) tuples `list`, the current state interval vector `box`, an interval extension of J_{ug}, $[J_{\text{ug}}]$, an interval extension of f_{g}, $[f_{\text{g}}]$, The RSP containing X^*_{k+1}, `targetPaving`, the node of the binary tree representing the RSP of X^*_{k+1} that corresponds to the current `node`, `controllerNode`.

output : The list, `list`, with its original parameters bisected and re-evaluated with sub-optimal parameters that are encountered removed.

```
begin
    originalElementsInList ⟵ listSize(list)
    if originalElementsInList = 0 then
        return
    if controllerNode = ∅ then
        listCostStar ⟵ +∞
    else if controllerNode.value ≠ INSIDE then
        listCostStar ⟵ +∞
    else if isLeaf(controllerNode) = TRUE then
        if listSize(controllerNode.caList) > 0 then
            listCostStar ⟵ controllerNode.caList.first.data.cost
        else
            listCostStar ⟵ +∞
    else
        listCostStar ⟵ controllerNode.cost
    while originalElementsInList > 0 do
        (φ, caCost) ⟵ takeTop(list)
        originalElementsInList ⟵ originalElementsInList − 1
        if caCost > listCostStar then
            continue
        (φ_L, φ_R) ⟵ bisect(φ, ·)
        controlActionCost ⟵ [J_ug](box, φ_L)
        if controlActionCost < listCostStar then
            nextState ⟵ [f_g](box, φ_L)
            (membershipValue, VNextState) ⟵ pavingMembershipAndCost(targetPaving, nextState)
            listCost ⟵ controlActionCost + VNextState
            if listCost < listCostStar then
                listCostStar ⟵ listCost
            if listCost ≤ listCostStar then
                addBottom((φ_L, listCost), list)
        controlActionCost ⟵ [J_ug](box, φ_R)
        if controlActionCost < listCostStar then
            nextState ⟵ [f_g](box, φ_R)
            (membershipValue, VNextState) ⟵ pavingMembershipAndCost(targetPaving, nextState)
            listCost ⟵ controlActionCost + VNextState
            if listCost < listCostStar then
                listCostStar ⟵ listCost
            if listCost ≤ listCostStar then
                addBottom((φ_R, listCost), list)
```

The `decimateLists` algorithm shown in Algorithm 29 removes all $(\phi, \texttt{cost})$ entries from the `caList` entries of the leaf nodes labelled `INSIDE` except, the entry with the lowest value of $\overline{\texttt{cost}}$. The `decimateList` function also returns the value of the $\overline{\texttt{cost}}$ of the remaining entry. The `listCost` entry of the aforementioned leaf node is also set to $\overline{\texttt{cost}}$.

Algorithm 29: The `decimateLists(node)` algorithm.

input : The root node, `node`, of a binary tree representing a regular subpaving.

output : The argument tree with the `caList` elements of each leaf node labelled `INSIDE` reduced to a single entry for each node and the `listCost` elements of the nodes set to the upper bound of the cost element of the said entry.

```
begin
    if isLeaf(node) = TRUE then
        if node.value = INSIDE then
            if listSize(node.caList) > 1 then
                node.listCost ⟵ decimateList(node.caList)
    else
        decimateLists(node.left)
        decimateLists(node.right)
```

The `calculateCosts` function shown in Algorithm 30 initializes the `cost` elements of the leaf nodes using the value of its `listCost` and by evaluating $[J_{\mathrm{x}}]$ for the state, `box`, associated with the current node.

Algorithm 30: The `calculateCosts(node, costPaving,` $[J_{\mathrm{x}}]$`)` algorithm.

input : The root node, `node`, of a binary tree that represents a regular subpaving, the root node of the existing controller paving, `costPaving`, and the interval extension of the state cost function, $[J_{\mathrm{x}}]$.

output : The argument tree with the `cost` elements of each leaf node labelled `INSIDE` set to its appropriate value.

```
begin
    if isLeaf(node) = TRUE then
        if node.listCost = NaN then
            node.listCost ⟵ costPaving.listCost

        if node.value = INSIDE then
            node.cost ⟵ node.listCost + [J_x](node.box)
        else
            node.cost ⟵ NaN
    else
        (costPavingLeft, costPavingRight) ⟵ returnChildNodes(costPaving)
        calculateCosts(node.left, costPavingLeft, [J_x])
        calculateCosts(node.right, costPavingRight, [J_x])
```

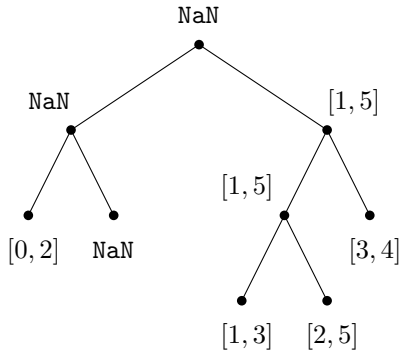


Figure 5-9: An example of a binary tree with the values of the `cost` elements of its leaf nodes backpropagated.

The `backPropCost` function shown in Algorithm 31 propagates the cost of the leaf nodes labelled `INSIDE` back to the root node. It does this by setting the cost of the intermediate nodes to the *interval hull* of the costs of its child nodes using the the *interval hull* operator, $\sqcup$, (see (2-34)) if both child nodes are labelled `INSIDE` and to `NaN` if this is not the case. The back-propagated values are used by Algorithm 32 to determine whether it is possible to discard one of the sub-trees based on the value of the `cost` cost entry of its root node.

An example of a binary tree with the value of the `cost` element[12] of each node displayed, is shown in Figure 5-9.

Algorithm 31: The `backPropCost(node)` algorithm.

input : The root node, `node`, of a regular subpaving.
output : The binary tree with the `cost` values of its leaf nodes back-propagated to its root node.

```
begin
    if isLeaf(node) = TRUE then
        if node.value = INSIDE then
            return node.cost
        else
            return NaN
    lCost ⟵ backPropCost(node.left)
    rCost ⟵ backPropCost(node.reft)
    if node.left.value = INSIDE and node.right.value = INSIDE then
        node.value ⟵ INSIDE
        node.cost ⟵ lCost ⊔ rCost
    else
        node.cost ⟵ NaN
    return node.cost
```

[12] Note that these are the `node.cost` elements, not the `cost` elements of the `caList` entries!

5-3-8 Set union

The `setUnionCost` algorithm shown in Algorithm 32 simultaneously implements the union operator, $\cup$, of (5-28), and the `merge` function of (5-29) and (5-30). Algorithm 32 is based on Algorithm 11 with additions for handling the additional information present in the nodes of the tree representing the regular subpaving.

Algorithm 32 consists of four parts:

- In the first part (the conditions on the lines marked (1a) to (3b) and their associated responses) it is checked whether it is possible to reject a (sub)tree based on the values of the `value` and `cost` entries of their root nodes. If the root node of one tree is labelled `INSIDE` and the root node of the other tree is labelled either `OUTISIDE` or `UNDETERMINED_OUTSIDE`, we can return a copy of the tree with the root node labelled `INSIDE` regardless of the contents of the other tree. Also if the value of the `cost` entry of the root node of one of the sub-trees has an upper bound that is strictly smaller than the lower bound of the other sub-tree, a copy of the tree can be returned. This is checked by the conditions on the lines marked (3a) and (3b).

- The second part (the conditions on the line marked (4) and its associated response) traverses the left and right sub-trees of both trees.

- The third part (the conditions on the lines marked (5) to (9) and their associated responses) handle the case when both nodes are leaf nodes. This part is a worked out version of the $\cup$ operator from Algorithm 11 with additional functionality for handling the additional node data. The most noticeable additions are the `mergeCaLists` and `cleanupList` functions that are used in the part that handles the case (on the line marked (6)) when both `value` entries are set to `INSIDE`. The `mergeCaLists` function creates a list containing copies of the entries of both its argument lists. For more information with regards to the `cleanupList` function the reader is referred to the discussion of Algorithm 17.

- The fourth part (the conditions on the lines marked (10) to (15) and their associated responses) handles the remaining cases (one of the nodes is a leaf node and one is an intermediate node). If the leaf node is labelled `OUTSIDE`, a copy of non-leaf node (and associated sub-tree) is returned; otherwise the same procedure is called for as when both nodes are intermediate nodes except that the now one of the nodes is 'virtually' traversed (as done in Algorithm 26 using Algorithm 27).

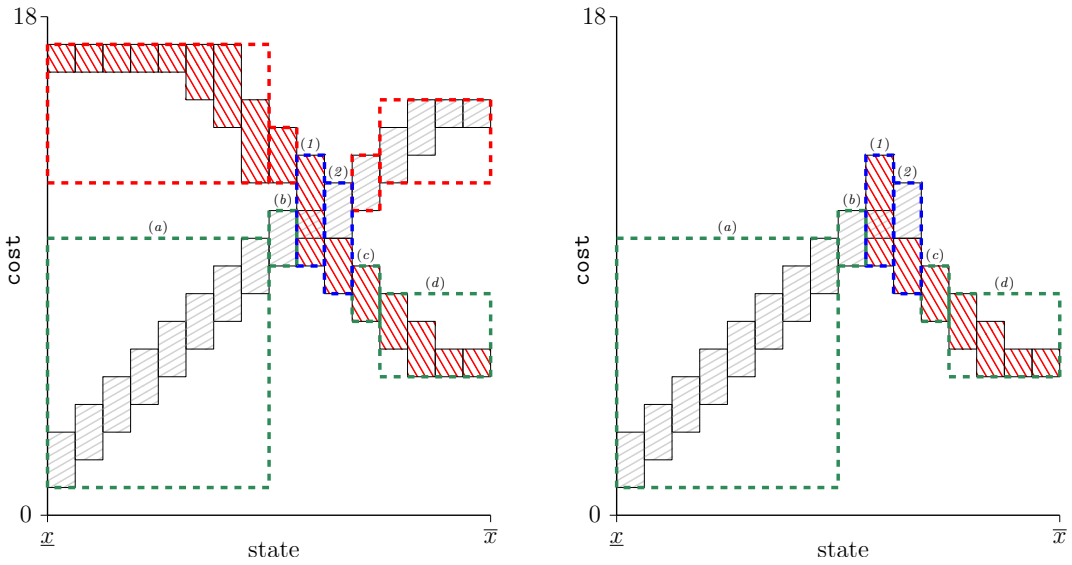
When the result has not yet been returned, then, on the line marked (16), the (potentially merged) result is returned.

Algorithm 32: The `setUnionCost(PavingIn1, PavingIn2)` function.

input : Two pavings `PavingIn1` and `PavingIn2` for which the union has to be determined.
output : A regular subpaving containing the union of `PavingIn1` and `PavingIn2`.

```
begin
(1a)  if PavingIn1.value = INSIDE and (PavingIn2.value = OUTSIDE or
      PavingIn2.value = UNDETERMINED_OUTSIDE) then
          return copyTreeCost(PavingIn1)
(1b)  if PavingIn2.value = INSIDE and (PavingIn1.value = OUTSIDE or
      PavingIn1.value = UNDETERMINED_OUTSIDE) then
          return copyTreeCost(PavingIn2)
(2)   if PavingIn1.value = INSIDE and PavingIn2.value = INSIDE and
      PavingIn1.cost ≠ NaN and PavingIn2.cost ≠ NaN then
(3a)      if overline(PavingIn1.cost) < overline(PavingIn2.cost) then
              return copyTreeCost(PavingIn1)
(3b)      else if overline(PavingIn1.cost) > overline(PavingIn2.cost) then
              return copyTreeCost(PavingIn2)
(4)   if isLeaf(PavingIn1) = FALSE and isLeaf(PavingIn2) = FALSE then
          PavingOut ⟵ newnode()
          PavingOut.left ⟵ setUnionCost(PavingIn1.left, PavingIn2.left)
          PavingOut.right ⟵ setUnionCost(PavingIn1.right, PavingIn2.right)
(5)   else if isLeaf(PavingIn1) = TRUE and isLeaf(PavingIn2) = TRUE then
(6)       if PavingIn1.value = INSIDE and PavingIn2.value = INSIDE then
              PavingOut ⟵ newnode()
              PavingOut.value ⟵ INSIDE
              PavingOut.caList ⟵ mergeCaLists(PavingIn1.caList, PavingIn2.caList)
              cleanupList(PavingOut.caList)
              return PavingOut
(7a)      else if PavingIn1.value = INSIDE then
              return copyNodeData(PavingIn1)
(7b)      else if PavingIn2.value = INSIDE then
              return copyNodeData(PavingIn2)
(8)       else if PavingIn1.value = OUTSIDE and PavingIn2.value = OUTSIDE then
              PavingOut ⟵ newnode()
              PavingOut.value ⟵ OUTSIDE
              return PavingOut
(9)       else
              PavingOut ⟵ newnode()
              PavingOut.value ⟵ UNDETERMINED
              return PavingOut
(10)  else if isLeaf(PavingIn1) = TRUE then
(11)      if PavingIn1.value = OUTSIDE then
              return copyTreeCost(PavingIn2)
(12)      else
              PavingOut ⟵ newNode()
              PavingOut.left ⟵ setUnionCost(PavingIn1, PavingIn2.left)
              PavingOut.right ⟵ setUnionCost(PavingIn1, PavingIn2.right)
(13)  else
(14)      if PavingIn2.value = OUTSIDE then
              return copyTreeCost(PavingIn1)
(15)      else
              PavingOut ⟵ newNode()
              PavingOut.left ⟵ setUnionCost(PavingIn1.left, PavingIn2)
              PavingOut.right ⟵ setUnionCost(PavingIn1.right, PavingIn2)
(16)  return reunite2(PavingOut)
```

An illustration of the use of the `cost` entries is given in Figures 5-10 and 5-11. Figure 5-10a



(a) A plot of the cost values of paving A (marked red) and paving B (marked grey).

(b) A plot of the cost values of the paving of $A \cup B$ after merger of the cost values.

Figure 5-10: An example of the input and output of the `setUnionCost` function.

shows a representation of the values of the `cost` entries of the nodes of binary trees representing two regular subpavings, with paving A marked red and paving B marked grey. Figure 5-10b shows the result of the merger of A and B.

The sections of the pavings marked with red dotted lines in Figures 5-10 and 5-11 correspond with the sub-pavings that can be rejected based on the value of the `cost` entry of their root nodes. The sections marked with green dotted lines correspond with the sub-pavings that can be accepted based on the value of the `cost` entry of their root nodes. These cases correspond with the lines marked (3a) and (3b) in Algorithm 32. The sections marked with blue dotted lines correspond to leaf nodes for which the values of the cost entries are ambiguous, i.e., neither value is strictly smaller or larger than the other.

The ambiguity can, for example, be resolved by only retaining the control action associated with the `cost` entry with the lowest upper bound, by applying[13] the `decimateLists` function (Algorithm 29) to the paving resulting from the merger of A and B.

Figure 5-11 shows the binary trees representing the regular subpavings of A, B, and $A \cup B$, and the values of the `cost` nodes. The sub-trees in Figure 5-11 marked red could, based on the value of the `cost` element of their root node be rejected since the corresponding root node of the other tree had a `cost` element that had a strictly lower value.

[13] It is assumed that the width of the value of the `cost` entries is sufficiently small to ensure that no controller parameters are discarded that, potentially, are significantly better than the one that will remain.

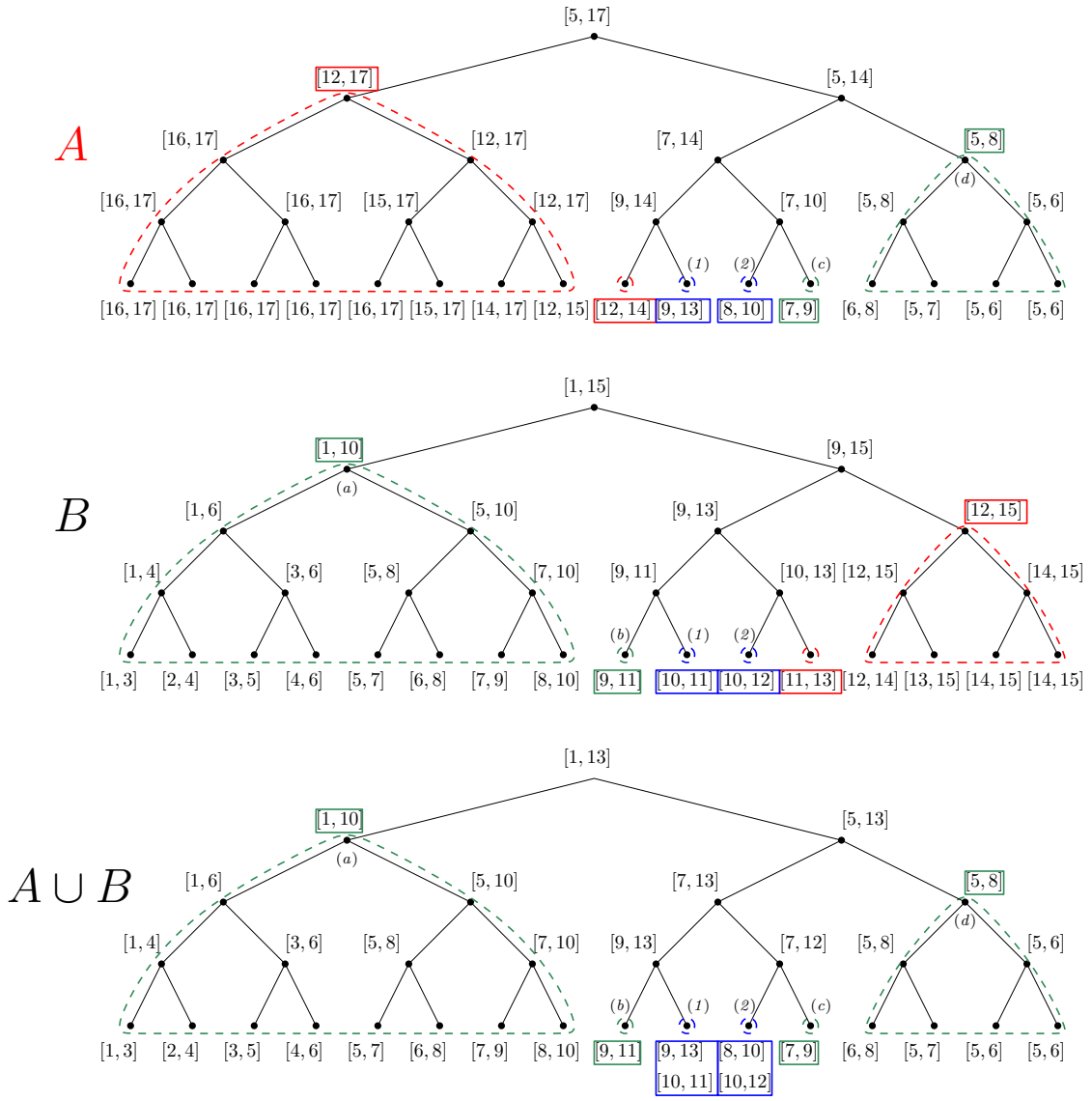


Figure 5-11: The binary trees representing the pavings shown in Figure 5-10.

5-4 Summary

In this chapter it has been shown how an algorithm can be constructed that, given a difference equation, a parametrised controller, an inequality constraint on the states and controller parameters, and a cost function, generates a partitioning of the state space, with for each section of the partitioning, a controller parameter such that a specified cost function is minimised.

First an idealised version of the algorithm is presented, using a number of idealised operations (like inversions, unions, intersections, and orthogonal projections of sets), and the minimisation of objective functions for sets of states. These operations are combined to create an algorithm that, for each state, can determine the controller parameters that minimises the specified cost function. The algorithm accomplishes this by solving the *Bellman equation* using *backward induction*.

Next a practical implementation of the idealised algorithm is presented. The practical implementation uses the algorithms and data structures introduced in Chapters 3 and 4 as the basis of practical implementations of the idealised operations presented in the previous section. Since these practical implementations are all based on interval methods, it is necessary that interval extensions (see Section 2-2) of the difference equation, cost function, parametrised controller and inequality can be constructed. As shown in Chapter 2 a sufficient condition for being able to construct an interval extension of a function is that the function can be evaluated using a finite number of evaluations of elementary real-valued operations (see Section 2-1), i.e., is a *closed-form* expression [5].

The implication[14] of this requirement is that the presented controller synthesis method can be applied for problems for which the difference equation, cost function, parametrised controller and inequality can be evaluated using a *closed-form* expression.

[14]This holds at least in theory. However, in Chapter 6 it will be shown that there are also a number of practical limitations that limit the use of the presented controller synthesis method.

Chapter 6

Benchmark problems

In this chapter the synthesis method described in Chapter 5 will be used to create controllers for two benchmark problems.

The aims of this chapter are to show that the synthesis method presented in this thesis can generate viable controllers for non-trivial control problems and how the implementation of the synthesis algorithm presented in Section 5-3 can be extended in order to handle a wider range of control problems than for which it was originally constructed.

Both benchmark problems are considered non-trivial in the sense that conventional approaches, such as creating linear regulators for the linearised plants, cannot be used.

Also an example is given of how a custom interval extension can be constructed that outperforms (less function evaluations and tighter solution bounds) the natural interval extensions (see Section 2-2-1) of the same function.

The first problem we consider is the generation of a controller for the *pendulum swing-up and stabilisation* benchmark problem, where the aim of the controller is to steer the angle of a pendulum to an upwards position and to keep it there, while the force of the motor that drives the pendulum is not sufficient to steer the pendulum directly to the desired position, thus requiring that the pendulum is swung back and forth until the momentum required to reach the desired position is obtained.

The second problem considered is the *car on a hill* benchmark problem, where a (frictionless) car has to be steered as fast as possible to a predefined target position and velocity by applying a horizontal force (that has an insufficient magnitude to steer the car directly to the desired position and velocity) while the car is positioned on a hill with a specified height profile.

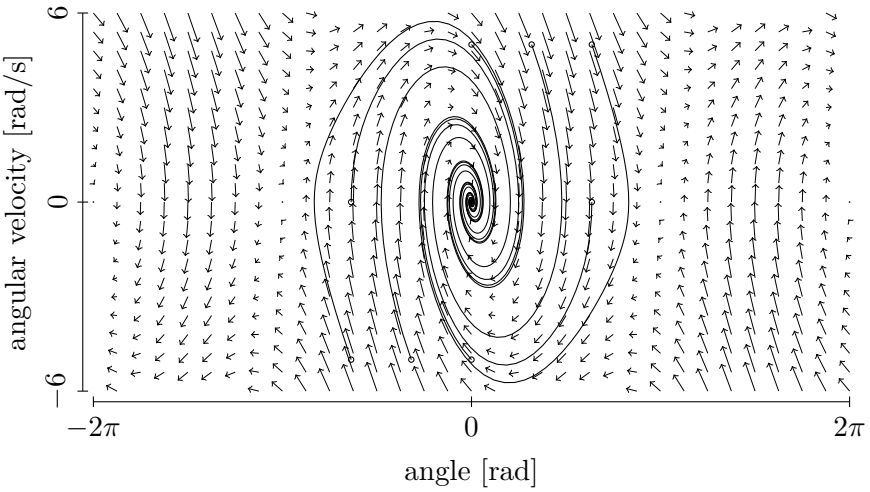


Figure 6-1: Phase plane and example trajectories of the uncontrolled pendulum.

6-1 Pendulum swing-up and stabilisation benchmark problem

The first benchmark problem for which a controller is generated is the pendulum swing-up and stabilisation problem. The control goal for this problem is to steer a rotational pendulum from any given angle θ and for rotational velocities $\dot{\theta}$ within a certain range, to a standstill in an upright position and to remain around this position by applying a torque to the rotational axis of the pendulum while satisfying constraints on the magnitude of the control action and rotational velocities while minimizing specified cost functions.

6-1-1 Modelling

The continuous-time (lumped) pendulum model and the values of its parameters are taken from [50]:

$$\ddot{\theta}(t) = \frac{1}{J}\left\{K_{\mathrm{m}}u(t) - mgL\sin\left(\theta(t)\right) - D\dot{\theta}(t)\right\} \tag{6-1}$$

with pendulum inertia $J = 0.005$ kg $\cdot$ m^2, motor gain $K_{\mathrm{m}} = 0.1$, mass $m = 0.1$ kg, $g = 9.81$ m $\cdot$ s^{-2}, pendulum length $L = 0.1$ m, and damping $D = 0.01$ kg $\cdot$ s^{-1}.

Figure 6-1 shows the phase plane of the uncontrolled ($u = 0$) pendulum with a number example trajectories (the initial states are marked using small circles).

As can be seen in Figure 6-1 all trajectories will converge to angles that are multiples of 2π and to angular velocities of zero[1].

The first step in the generation of the controller is to generate a discrete-time approximation of the continuous-time dynamics of the pendulum. The most straight forward approach for

[1]For the example trajectories the angles converge to zero; other trajectories converge to other multiples of 2π but may fall outside the range of the plot.

doing this is the *Forward Euler* method (see [6], Section 2.3), which results in the following expression of the discrete-time dynamics:

$$\theta(k+1) = \theta(k) + t_{\mathrm{s}}\dot{\theta}(k) \tag{6-2}$$

$$\dot{\theta}(k+1) = \dot{\theta}(k) + t_{\mathrm{s}} \underbrace{\frac{1}{J}\Big\{K_{\mathrm{m}}u - mgL\sin\big(\theta(k)\big) - D\dot{\theta}(k)\Big\}}_{\ddot{\theta}(k)} \tag{6-3}$$

The advantage of the Forward Euler method is its simplicity, but it has as an disadvantage that the (global) approximation error is roughly proportional to the step size for small step sizes.

Another approach that can be used the generate a discrete-time model is *Heun's method* (see [6], Section 3.1), which results in the following expression of the discrete-time dynamics:

$$\tilde{\theta}(k+1) = \theta(k) + t_{\mathrm{s}}\dot{\theta}(k) \tag{6-4}$$

$$\tilde{\dot{\theta}}(k+1) = \dot{\theta}(k) + t_{\mathrm{s}}\ddot{\theta}(k) \tag{6-5}$$

with $\ddot{\theta}(k)$ as specified in (6-3), and

$$\theta(k+1) = \theta(k) + \frac{t_{\mathrm{s}}}{2}\big(\dot{\theta}(k) + \tilde{\dot{\theta}}(k+1)\big) \tag{6-6}$$

$$\dot{\theta}(k+1) = \dot{\theta}(k) + \frac{t_{\mathrm{s}}}{2}\big(\ddot{\theta}(k) + \tilde{\ddot{\theta}}(k+1)\big) \tag{6-7}$$

with $\tilde{\ddot{\theta}}(k+1)$ defined as:

$$\tilde{\ddot{\theta}}(k+1) = \frac{1}{J}\Big\{K_{\mathrm{m}}u - mgL\sin\big(\tilde{\theta}(k+1)\big) - D\tilde{\dot{\theta}}(k+1)\Big\} \tag{6-8}$$

The advantage of Heun's method over the Forward Euler method is that its (global) approximation error is roughly proportional to the square of the step size for small time step sizes, but it has as disadvantage that it requires two evaluations of the dynamics per time step. The second consideration in the creation of a discrete-time approximation of a continuous-time process is the choice of the time step. Since the controller is generated for the discrete-time process, the sampling time of the controller is directly related to the step time used by the discrete-time approximation of the continuous-time process. In the current benchmark problem we choose a time step that results in a good approximation of the continuous-time behaviour of the plant model when used with the chosen integration method.

Figure 6-2 shows the simulated angles and angular velocities of the pendulum when using the *Dormand-Prince* method[2], Heun's method with a time step of 0.1 seconds, and the Forward Euler method with sampling times of 0.1 and 0.05 seconds.

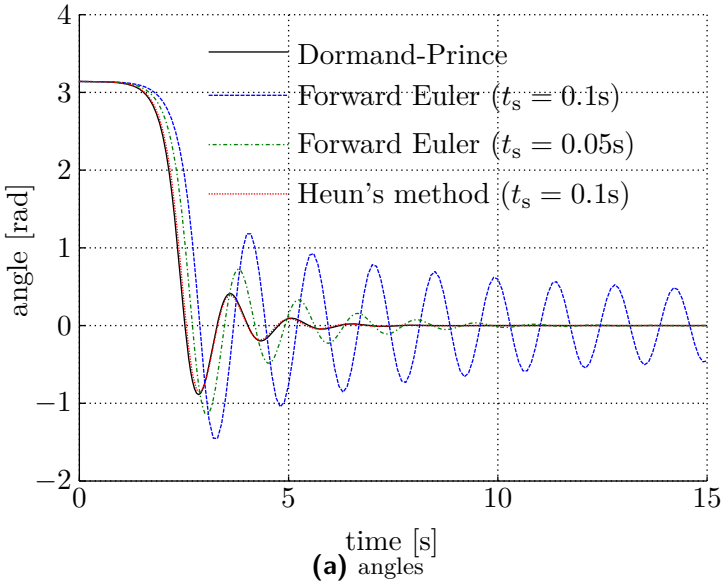
The results of the Dormand-Prince method are shown as a reference and are considered to be equivalent, for all practical purposes, to the continuous-time responses for this example. It should be obvious that the approximation using the Forward Euler method using a time step of 0.1 seconds is useless, while the approximation using Heun's method with a time step of 0.1 seconds is nearly indistinguishable from the approximation using the Dormand-Prince method.[3]

[2]The Dormand-Prince method is the default method in Matlab's ODE solver.

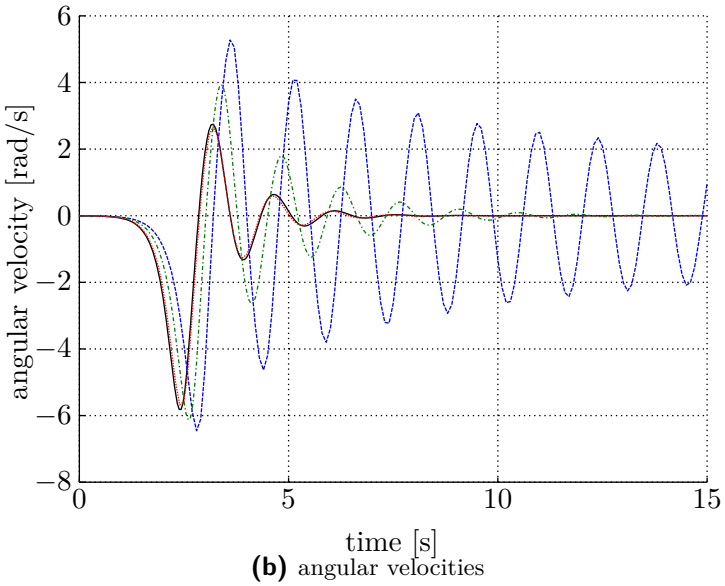
[3]This is still the case when values other than zero are used for the control action u.

Since Heun's method uses twice the number of evaluations of the dynamics as the Forward Euler method it should be fair to consider what the effect of halving the time step of the Forward Euler method are (so that the number of evaluations of the dynamics becomes equal to that of Heun's method). While the results of the Forward Euler using a time step of 0.05 seconds are obviously better than when using a time step of 0.1 seconds, they are still noticeably worse than Heun's method using a time step of 0.1 seconds.

Based on these results we choose to use Heun's method for the construction of our discrete-time pendulum model. While even more accurate simulation methods exist (see [6]) only Heun's method is considered as an alternative to the Forward Euler method because at this point it is deemed sufficiently accurate (at a sample time of 0.1 seconds it is almost indistinguishable from Dormand-Prince) and still simple enough (it allows the discrete-time model to be expressed as a closed-form expression and the number of dependencies are kept to a minimum compared with multi-step methods based on the *Runge-Kutta* method).



(a) angles



(b) angular velocities

Figure 6-2: Behaviour of the uncontrolled pendulum models for $\theta_0 = \pi - 0.01$, $\dot{\theta}_0 = 0$, $u = 0$.

6-1-2 Control goals

As mentioned in the introduction of this section, the goal of the controller is to steer the angle θ and angular velocity $\dot{\theta}$ of the pendulum into a specified range of angles Θ_{target} and angular velocities $\dot{\Theta}_{\text{target}}$ and to remain there. The target angles and angular velocities, Θ_{target} and $\dot{\Theta}_{\text{target}}$, are specified as:

$$\Theta_{\text{target}} = \bigcup_{n \in \mathbb{Z}} [2\pi n + \pi - 0.05,\ 2\pi n + \pi + 0.05] \tag{6-9}$$

$$\dot{\Theta}_{\text{target}} = [-0.05,\ 0.05] \tag{6-10}$$

The aim is to generate a controller that achieves the aforementioned goal for the following ranges of angles and angular velocities:

$$-\infty \leq \theta \leq +\infty \tag{6-11}$$

$$-6 \leq \dot{\theta} \leq 6 \tag{6-12}$$

with a control action that is limited to the range:

$$-0.8 \leq u \leq 0.8 \tag{6-13}$$

and minimizing the cumulative sum of the following cost functions:

$$J_{\text{x}}(\theta, \dot{\theta}) = (\theta \bmod 2\pi - \pi)^2 + 0.1\,\dot{\theta}^2 \tag{6-14}$$

$$J_{\text{u}}(u) = 0.05\,u^2 \tag{6-15}$$

until the target set is reached.

6-1-3 Controller synthesis

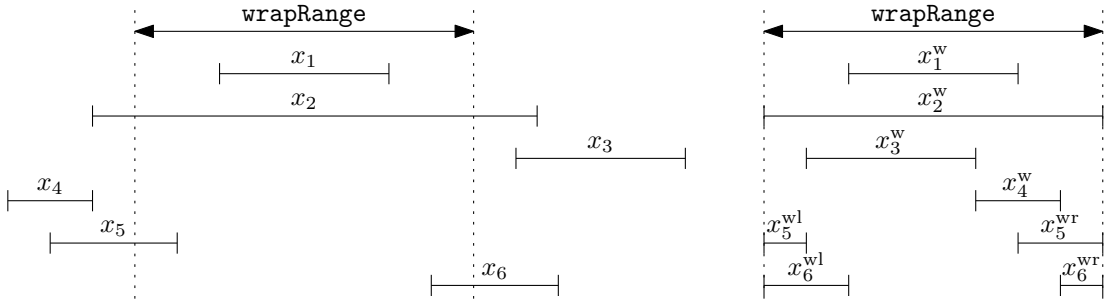
Before we can start defining the arguments of the controller synthesis algorithm we have to handle the infinite number of target angles (6-9), infinite range of θ (6-11), and the occurrence of the modulo operation in (6-14). The first two of these issues can be handled by mapping the unconstrained range of θ to $[0, 2\pi]$, by using $\theta \bmod 2\pi$ rather than θ in our calculations. This is allowed since the only occurrence of θ in our dynamics is inside a sine function and $\sin(\theta) = \sin(\theta \bmod 2\pi)$.

Due to the use of the modulo operator we can now suffice by generating the controller for the range:

$$0 \leq \theta \leq 2\pi \tag{6-16}$$

The occurrence of the modulo operation in (6-14) can now be disregarded since the range of θ is now limited to $[0, 2\pi]$.

The use of the modulo operation requires some attention since it is used with intervals rather than real-valued values.



(a) Original intervals. **(b)** Wrapped intervals.

Figure 6-3: Original intervals and their wrapped versions.

A generalised version (for wrapping arbitrary ranges) of the modulo operation is implemented by substituting the `pavingMembershipAndCost` functions in Algorithms 19 and 28 by `pavingMembershipAndCostWrap`.

We now show how the *wrapper function* `pavingMembershipAndCostWrap` for `pavingMembershipAndCost` (Algorithm 20) is used to map the range $(-\infty, +\infty)$ to an interval specified by `wrapRange`.

For a given interval we can encounter the following cases:

- x_1: The interval is located within `wrapRange`, no wrapping is required.
- x_2: The width of the interval exceeds the width of `wrapRange`, the wrapped interval is set to `wrapRange`. Note that x_2 can also be located outside `wrapRange`.
- x_3: The lower bound of the interval is equal to or exceeds the upper bound of `wrapRange`, the width of `wrapRange` is subtracted from the interval. After adjustment of the interval the wrapping function is called again for the new interval, until a case like x_1, x_5, or x_6 occurs[4].
- x_4: The upper bound of the interval is equal to or lower than the upper bound of `wrapRange`, the width of `wrapRange` is added to the interval. As with x_3 the wrapping function is called again until a case like x_1, x_5, or x_6 occurs.
- x_5: The interval intersects the lower bound of `wrapRange`, the interval is split in two parts, with the width of `wrapRange` added to the interval outside `wrapRange`.
- x_6: The interval intersects the upper bound of `wrapRange`, the interval is split in two parts, with the width of `wrapRange` subtracted from the interval outside `wrapRange`.

These cases, and their resulting intervals, are illustrated in Figure 6-3.

The peudo-code for the `pavingMembershipAndCostWrap` function is shown in Algorithm 33 where the lines that check the various cases of x_n are marked with (n).

[4]This approach is used instead of using the modulo operator `%` because the modulo operator in C(++) does not actually calculate the *modulus* but the *remainder* (see Section A7.6 of [20]). This causes problems when using the operator with negative values.

Algorithm 33: The pavingMembershipAndCostWrap(node, X, wrapAxis, wrapRange) function.

input : The root node of a regular subpaving, node, a box, $X \in \mathbb{IR}^n$, the current recursion depth, depth, the wrapping axis wrapAxis, and the wrapping range wrapRange.

output : The membership of the wrapped box X with respect to the set approximated by the paving with root node node, and the lowest lower bound and the highest upper bound, of all nodes labelled INSIDE intersecting with X.

begin

$x \longleftarrow X_{\texttt{wrapAxis}}$

(1) **if** $\underline{x} \geq \underline{\textit{wrapRange}}$ *and* $\overline{x} \leq \overline{\textit{wrapRange}}$ **then**

return pavingMembershipAndCost(node, X, 0)

(2) **if** $w(x) \geq w(\textit{wrapRange})$ **then**

wrappedBox $\longleftarrow X$

$\texttt{wrappedBox}_{\texttt{wrapAxis}} \longleftarrow$ wrapRange

return pavingMembershipAndCost(node, wrappedBox, 0)

(3) **if** $\underline{x} \geq \overline{\textit{wrapRange}}$ **then**

wrappedBox $\longleftarrow X$

$\texttt{wrappedBox}_{\texttt{wrapAxis}} \longleftarrow X_{\texttt{wrapAxis}} - w(\texttt{wrapRange})$

return pavingMembershipAndCostWrap(node, wrappedBox, wrapAxis, wrapRange)

(4) **if** $\overline{x} \leq \underline{\textit{wrapRange}}$ **then**

wrappedBox $\longleftarrow X$

$\texttt{wrappedBox}_{\texttt{wrapAxis}} \longleftarrow X_{\texttt{wrapAxis}} + w(\texttt{wrapRange})$

return pavingMembershipAndCostWrap(node, wrappedBox, wrapAxis, wrapRange)

if $\underline{x} < \underline{\textit{wrapRange}}$ *or* $\overline{x} > \overline{\textit{wrapRange}}$ **then**

lBox $\longleftarrow X$

rBox $\longleftarrow X$

(5) **if** $\underline{x} < \underline{\textit{wrapRange}}$ **then**

$\texttt{lBox}_{\texttt{wrapAxis}} \longleftarrow [\underline{\texttt{wrapRange}},\ \overline{x}]$

$\texttt{rBox}_{\texttt{wrapAxis}} \longleftarrow [\underline{x} + w(\texttt{wrapRange}),\ \overline{\texttt{wrapRange}}]$

(6) **else**

$\texttt{lBox}_{\texttt{wrapAxis}} \longleftarrow [\underline{\texttt{wrapRange}},\ \overline{x} - w(\texttt{wrapRange})]$

$\texttt{rBox}_{\texttt{wrapAxis}} \longleftarrow [\underline{x},\ \overline{\texttt{wrapRange}}]$

(lMembership, lCost) $\longleftarrow$ pavingMembershipAndCost(node, lBox, 0)

(rMembership, rCost) $\longleftarrow$ pavingMembershipAndCost(node, rBox, 0)

if *lMembership* $=$ *INSIDE* *and* *rMembership* $=$ *INSIDE* **then**

return (INSIDE, lCost $\sqcup$ rCost)

if *lMembership* $=$ *OUTSIDE* *and* *rMembership* $=$ *OUTSIDE* **then**

return (OUTSIDE, NaN)

return (UNDETERMINED, NaN)

For the current benchmark problem `wrapRange` is set to $[0, 2\pi]$.

For the parametrisation of the controller (5-3) we use a straightforward implementation:

$$g\Big(x(k), \Phi\big(x(k)\big)\Big) = \Phi\big(x(k)\big) \tag{6-17}$$

We can now define the arguments of the controller synthesis algorithm (Algorithm 14).

The constraints on the angular velocity and control action can be implemented by setting the `box` argument of the synthesis algorithm to the allowed ranges:

$$\texttt{box} \longleftarrow \begin{bmatrix} [0, 2\pi] \\ [-6.0, 6.0] \\ [-0.8, 0.8] \end{bmatrix} \tag{6-18}$$

This is allowed since the constraints are *box constraints*[5], see the description of line (1a) of the synthesis algorithm, Algorithm 14, in Section 5-3-3. The effect of this approach is that angular velocities and control actions outside `box` are not considered during controller synthesis[6], thereby implicitly satisfying the constraints. We are allowed to consider only the angles inside the range $[0, 2\pi]$ since all other angles are mapped to this range.

J_{ug} (see (5-11)) is constructed by substitution of (6-17) in (6-15), f_{g} (see (5-9)) is constructed in the same fashion by substitution of (6-17) in (6-3) and (6-8).

For f_{g} we use a sampling time of 0.03 seconds. This value was chosen since it results in a good approximation of the 'real' system[7] by the discrete-time model[8] while limiting the number of iterations required to generate the controller to a reasonable amount. It must be noted that using a different sampling time, e.g., one of 0.1 seconds will still result in a controller that satisfies all requirements.

We now specify the `iterations` parameter, which specifies the number of iterations that the synthesis algorithm is run and which should be at least be as large as the maximum number of time steps it takes to reach the target set from any given state in the range of states for which the controller is generated. The problem is that this quantity is hard to determine a priori. One of the ways to circumvent this problem is not to specify the maximum number of iterations at forehand but to use other stopping criteria, for example by monitoring whether the generated controller covers all the states for which the controller is generated or by monitoring the increase of the volume covered by the controller at each iteration.

For the current benchmark problem the synthesis loop is terminated when all states are covered. This is checked by monitoring the back-propagated value of the `cost` entry of the root node of the binary tree representing the regular sub-paving of the state space. Note that

[5] I.e. the constraints only consist of lower and upper bounds.

[6] This due to the fact that during the determination of the joint range of states and controller parameters that can reach the previous set of backwards reachable states (5-25), all controller parameters that can steer the next state outside region of the state space for which the controller is generated, are rejected. In other words the joint range of the states and control actions is implicitly constrained to $f_{\text{g}}^{-1}(X_{\text{range}})$, where X_{range} is the region of the state space for which the controller is generated. This property is also mentioned in the footnote of the text accompanying (5-15).

[7] The continuous-time model simulated using the Dormand-Prince method.

[8] Using Heun's method.

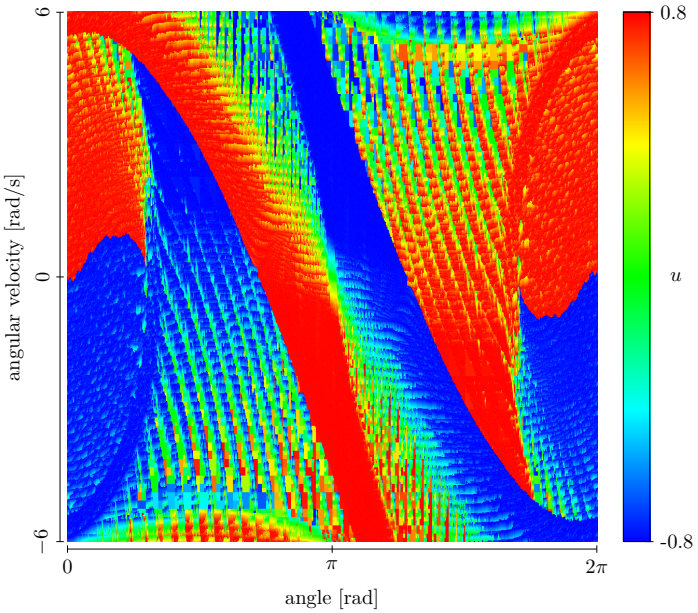


Figure 6-4: Heat map of the control actions.

this approach cannot be used when the controller cannot cover the entire specified region of the state space, as shall be shown in Section 6-2.

What remains is defining the `parameterScheduler` function (see the description of line (0f) of the synthesis algorithm). For the current benchmark problem `parameterScheduler` always returns $(30, 10)$.

We now have all functions and parameters required to synthesize the controller. For the current benchmark problem the synthesis algorithm terminates after 73 iterations[9].

Figures 6-4 and 6-5 show heat maps of respectively the control action, u, and the value function, V, for each state in the range for which the controller is generated. Figure 6-5 also shows the phase plane of the closed-loop system.

6-1-4 Closed-loop system

The closed-loop behaviour of the controller plant interconnection is simulated using the Dormand-Prince method. As in Section 6-1-1 the plant model is simulated using Dormand-Prince; this is assumed to be a reasonable approximation of the continuous-time behaviour of the plant.

Figure 6-6 shows a number of trajectories of the closed-loop system for different initial states. The initial states are marked with small circles, the target sets are indicated with the boxes at $(-\pi, 0)$ and $(\pi, 0)$. As a background the heat map of V is used to give an impression of the different regions in the state space.

[9]These take around 860 minutes of computation time on a Macbook Pro with a 2.9 GHz Intel Core i7 processor.

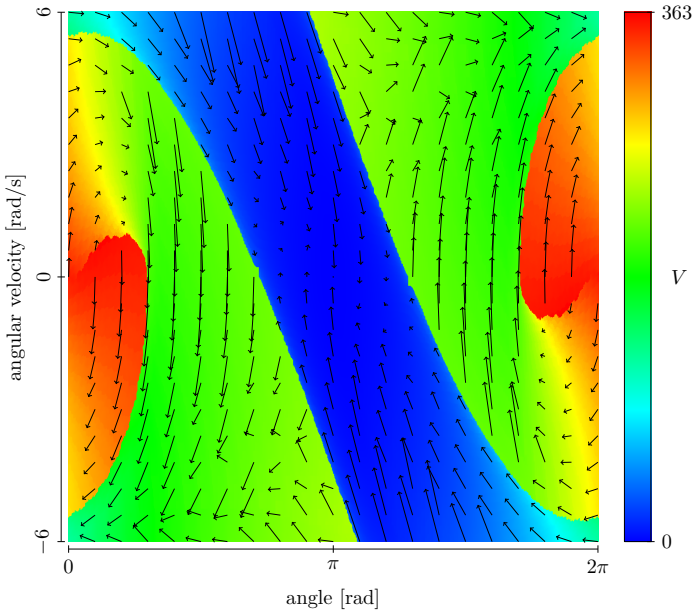


Figure 6-5: Phase plane of the closed-loop system and heat map of the value function V.

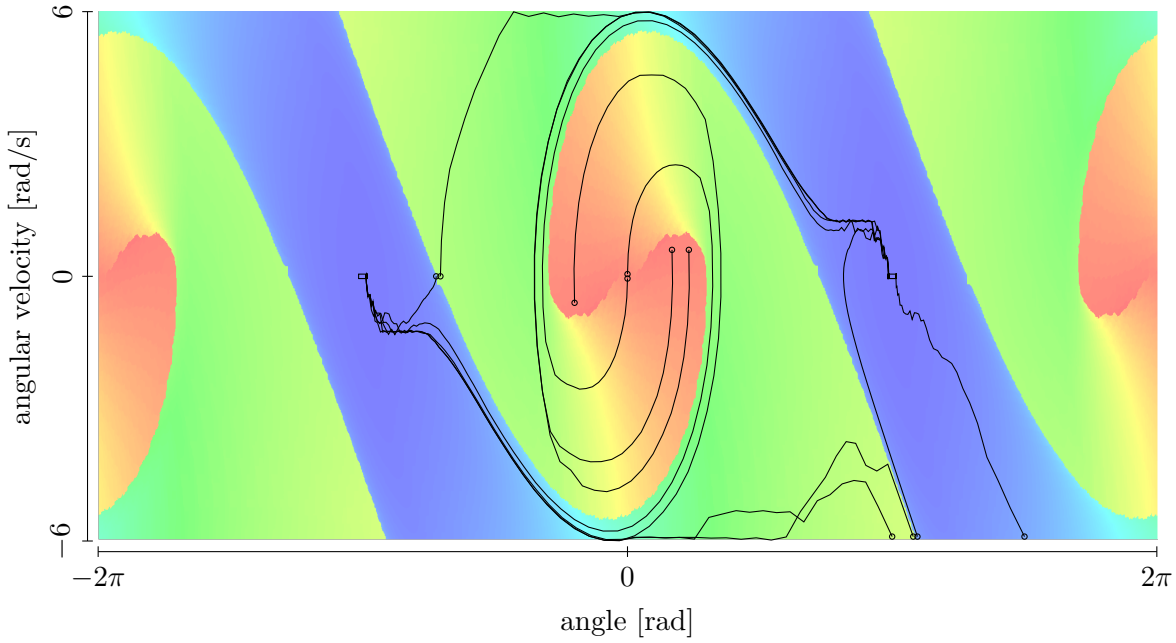


Figure 6-6: Example trajectories of the closed-loop system.

Figure 6-7 shows a typical response of the closed-loop system, with the initial angle and angular velocity set to zero. The dashed lines are used to indicate the target sets in the top sub-figure and bounds on the control action in the bottom sub-figure. The oscillatory behaviour after $t = 2.5$ seconds is due to the controller correcting the angular velocity to prevent it from drifting outside the target set. This behaviour can be prevented by handling the control of the pendulum over to, e.g., a linear quadratic regulator when the target set is

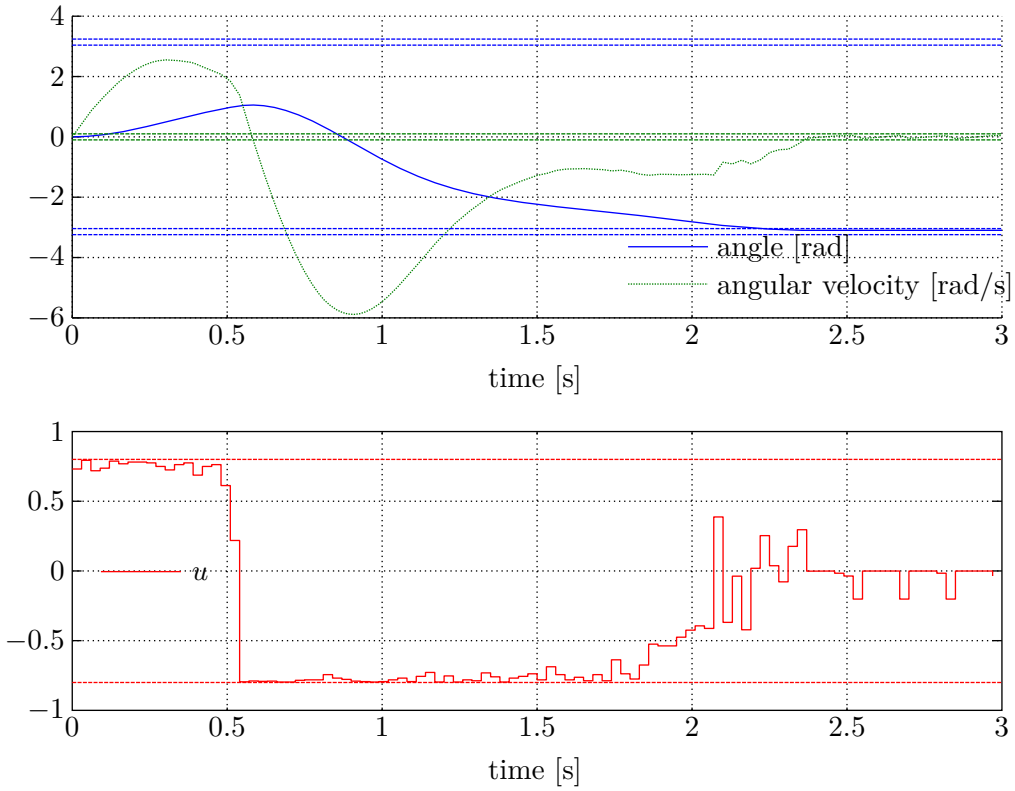


Figure 6-7: Closed-loop response and control action of the pendulum benchmark for $\theta_0 = 0$, $\dot{\theta}_0 = 0$. The dashed lines are used to indicate the target sets in the top sub-figure and bounds on the control action in the bottom sub-figure.

reached.

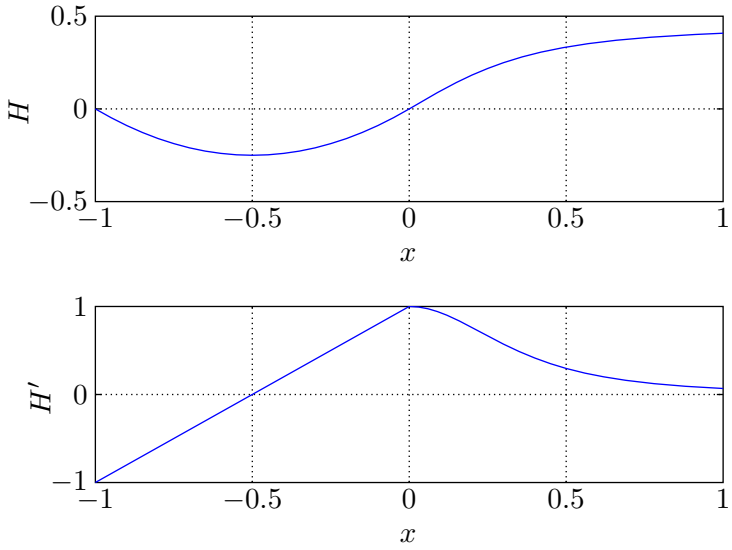


Figure 6-8: Height and slope.

6-2 Car on the hill benchmark problem

The second benchmark problem we consider is the *Car on the hill* problem as described in Appendix A.2 of [32][10].

The aim in this benchmark problem is to steer a (frictionless) car as fast as possible from from any initial velocity $\dot{x}$ and position x within a certain range, to a predefined target position and velocity range by applying a horizontal force a to the car.

6-2-1 Modelling

As in [32] no units are used with our model. The height of the terrain along which the car moves is defined by $H(x)$:

$$H(x) = \begin{cases} x^2 + x & \text{if } x < 0 \\ x/\sqrt{1+5x^2} & \text{if } x \geq 0 \end{cases} \tag{6-19}$$

and its slope $H'(x)$:

$$H'(x) \stackrel{\text{def}}{=} \frac{d}{dx}H(x) = \begin{cases} 2x+1 & \text{if } x < 0 \\ 1/\left(\left(1+5x^2\right)^{2/3}\right) & \text{if } x \geq 0 \end{cases} \tag{6-20}$$

Figure 6-8 shows plots of H and H'. The car's dynamics are given as:

$$\ddot{x} = \underbrace{\underbrace{\frac{a}{M\sqrt{1+\left(H'(x)\right)^2}}}_{\text{xdd}_1} \underbrace{-\frac{gH'(x)}{1+\left(H'(x)\right)^2}}_{\text{xdd}_2}}_{\text{xdd}} \tag{6-21}$$

[10] In that paper it is referred to as *The puck on the hill*.

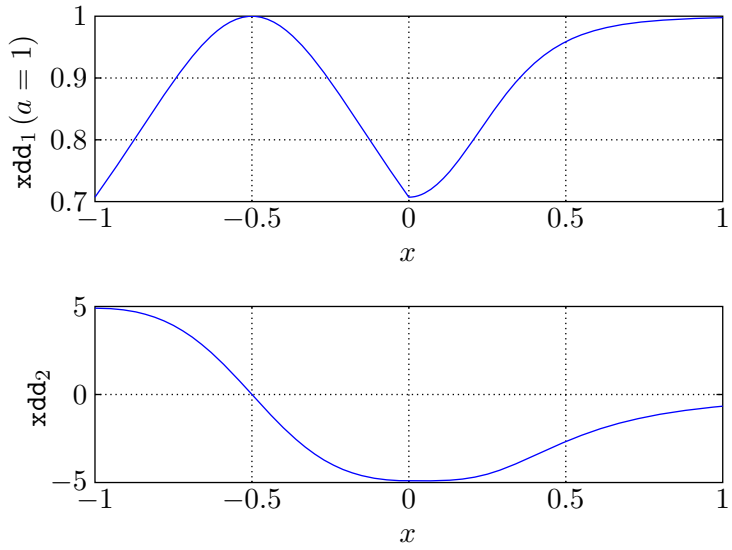


Figure 6-9: $\texttt{xdd}_1$ and $\texttt{xdd}_2$

In order to use (6-21) with the controller synthesis algorithm we need to create an interval extension of it. The most straightforward way of doing this would be to create the *natural interval extension* (see Subsection 2-2-1), and while this generates a valid interval extension of (6-21) the quality of the resulting interval extension would be rather bad due to the multiple occurrences of $H'(x)$ in (6-21), which, due to the *loss of dependency* (see Subsection 2-3-1), causes significant overestimation of the true range of the resulting interval.

Another approach is to generate a custom interval extension of (6-21). In the custom interval extension that is created two properties of interval extensions are used: the first property is that interval extensions of real-valued functions that are either strictly non-increasing or non-decreasing, can be constructed using only endpoint evaluations (see accompanying text of (2-8) and (2-9)); the second property is that we can split an argument interval into multiple smaller intervals, evaluate these intervals using the interval extension, and merge the results using the (interval) union operator (see Section 2-3-3).

If we look at Figure 6-9 we can see that $\texttt{xdd}_1$ can be split-up into three segments that are either strictly non-increasing or strictly non-decreasing, and $\texttt{xdd}_2$ can be split-up into two segments. The pseudo code of the custom interval extension[11] [xdd] of (6-21) xdd is shown in Algorithm 34. In the first part of the algorithm we (recursively) divide the interval-valued input arguments, a and x, into strictly non-decreasing or non-increasing segments and merge the results of their evaluation. In the second part of the algorithm the intervals generated by the first part of the algorithm are evaluated using evaluation of their endpoints. The results generated by the custom interval extension are significantly better, i.e., their bounds are tighter than those generated by evaluation of the same arguments by the natural interval

[11] Note that this is not a *reliable* interval extension, i.e., it does not take the rounding errors of the floating point operations into account, this in contrast to natural interval extensions constructed using one of the interval libraries (like [45] or [24]). This not an issue for this case study since the errors introduced by the modelling and simulation are likely to be multiple orders of magnitude larger than those introduced by the floating point operations.

extension of the same real-valued function.

Algorithm 34: Custom interval extension of `xdd`, [`xdd`].

input : Position interval $x \in \mathbb{IR}$, control action interval $a \in \mathbb{IR}$.
output : Acceleration interval $\ddot{x} \in \mathbb{IR}$.

begin
  if $\underline{a} < 0$ *and* $\overline{a} > 0$ **then**
    return $[\texttt{xdd}](x, [\underline{a}, 0]) \sqcup [\texttt{xdd}](x, [0, \overline{a}])$
  if $\underline{x} < -0.5$ *and* $\overline{x} > -0.5$ **then**
    return $[\texttt{xdd}]([\underline{x}, -0.5], a) \sqcup [\texttt{xdd}]([-0.5, \overline{x}], a)$
  if $\underline{x} < 0$ *and* $\overline{x} > 0$ **then**
    return $[\texttt{xdd}]([\underline{x}, 0], a) \sqcup [\texttt{xdd}]([0, \overline{x}], a)$
  if $\overline{a} \leq 0$ **then**
    if $\overline{x} \leq -0.5$ **then**
      return $\left[\texttt{xdd}(\overline{x}, \underline{a}),\ \texttt{xdd}(\underline{x}, \overline{a})\right]$
    else if $\underline{x} \geq -0.5$ *and* $\overline{x} \leq 0$ **then**
      return $\left[\mathrm{xdd}_1(\underline{x}, \underline{a}) + \mathrm{xdd}_2(\overline{x}),\ \mathrm{xdd}_1(\overline{x}, \overline{a}) + \mathrm{xdd}_2(\underline{x})\right]$
    else if $\underline{x} \geq 0$ **then**
      return $\left[\mathrm{xdd}_1(\overline{x}, \underline{a}) + \mathrm{xdd}_2(\underline{x}),\ \mathrm{xdd}_1(\underline{x}, \overline{a}) + \mathrm{xdd}_2(\overline{x})\right]$
  else
    if $\overline{x} \leq -0.5$ **then**
      return $\left[\mathrm{xdd}_1(\underline{x}, \underline{a}) + \mathrm{xdd}_2(\overline{x}),\ \mathrm{xdd}_1(\overline{x}, \overline{a}) + \mathrm{xdd}_2(\underline{x})\right]$
    else if $\underline{x} \geq -0.5$ *and* $\overline{x} \leq 0$ **then**
      return $\left[\texttt{xdd}(\overline{x}, \underline{a}),\ \texttt{xdd}(\underline{x}, \overline{a})\right]$
    else if $\underline{x} \geq 0$ **then**
      return $\left[\texttt{xdd}(\underline{x}, \underline{a}),\ \texttt{xdd}(\overline{x}, \overline{a})\right]$

An example of the improvement that is obtained by using the custom interval extension of `xdd` with respect to its natural interval extension is shown in Figures 6-10 and 6-11. These respectively show the result of the evaluation of the natural interval extension[12] $[\texttt{xdd}]_{\mathrm{n}}$ and the custom interval extension [`xdd`] for $x = \{[-1, -0.6], [-0.6, -0.2], [-0.2, 0.2], [0.2, 0.6], [0.6, 10.0]\}$ and $a = [-1, 1]$. In both plots the result of the evaluation[13] of $\texttt{xdd}(x, a)$ for $x = [-1, 1]$ and $a = [-1, 1]$ are shown in red.

Figure 6-12 shows the phase plane of the uncontrolled system together with a number of example trajectories. The square indicates the range of the states for which the controller is generated. We use the discrete-time approximation of dynamics as described in [32]:

$$x(k+1) = x(k) + t_{\mathrm{s}}\dot{x}(k) + \tfrac{1}{2}t_{\mathrm{s}}^2\ddot{x}(k) \tag{6-22}$$
$$\dot{x}(k+1) = \dot{x}(k) + t_{\mathrm{s}}\ddot{x}(k) \tag{6-23}$$

with $t_{\mathrm{s}} = 0.01\mathrm{s}$.

[12] We use the subscript n to differentiate the natural extension from the custom extension.
[13] This evaluation is approximated by evaluating `xdd` for a large number of different values of x and a in the ranges $[-1, 1]$ and $[-1, 1]$ respectively and plotting the resulting responses.

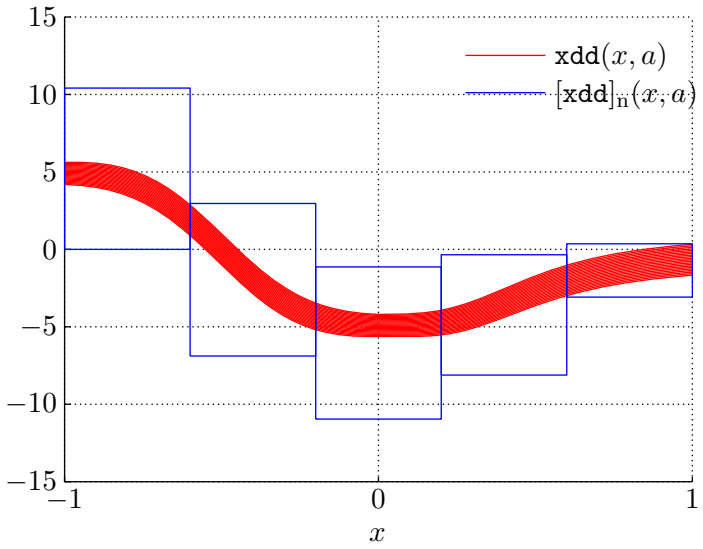


Figure 6-10: Examples of the evaluation of the natural interval extension $[\texttt{xdd}]_{\text{n}}$ for $x = \{[-1, -0.6], [-0.6, -0.2], [-0.2, 0.2], [0.2, 0.6], [0.6, 10.0]\}$ and $a = [-1, 1]$ and the true range xdd.

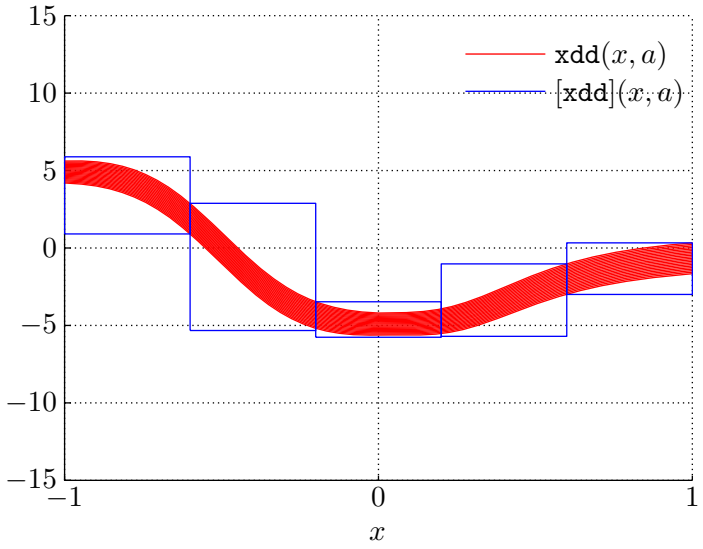


Figure 6-11: Examples of the evaluation of the custom interval extension $[\texttt{xdd}]$ for $x = \{[-1, -0.6], [-0.6, -0.2], [-0.2, 0.2], [0.2, 0.6], [0.6, 10.0]\}$ and $a = [-1, 1]$ and the true range xdd.

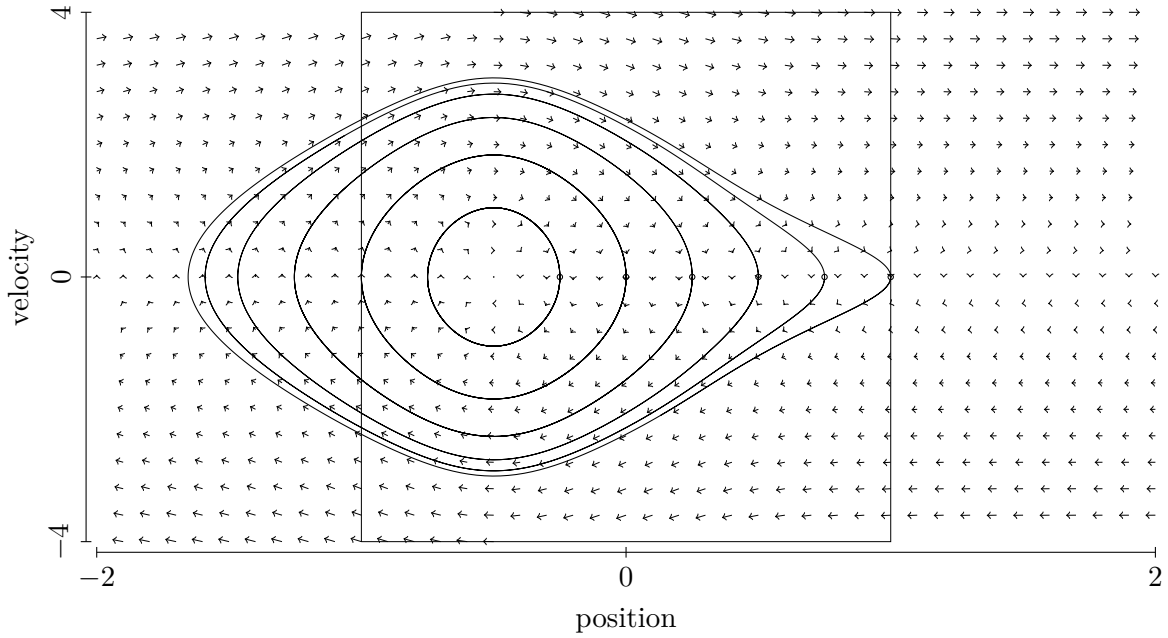


Figure 6-12: Example trajectories unregulated system and range controller.

6-2-2 Control goals

The target states are specified as:

$$x_{\text{target}} = [0.5, 0.7] \tag{6-24}$$
$$\dot{x}_{\text{target}} = [-0.1, 0.1] \tag{6-25}$$

we aim to create a controller for the following range of states:

$$-1 \leq x \leq 1 \tag{6-26}$$
$$-4 \leq \dot{x} \leq 4 \tag{6-27}$$

Where the velocity range of [35] is used[14]. The range of the control action is specified as:

$$-4 \leq a \leq 4 \tag{6-28}$$

The cost functions J_{x} and J_{u} are specified as:

$$J_{\text{x}}(x) = \begin{cases} 0 & \text{if } x \in X_{\text{target}} \\ 1 & \text{otherwise} \end{cases} \tag{6-29}$$
$$J_{\text{u}}(u) = 0.1\,|u| \tag{6-30}$$

The aim of the state cost function J_{x} is to generate a controller that steers the state to the target states in the shortest time, while the control action cost function J_{u} aims to minimize the absolute value of the control actions.

6-2-3 Controller synthesis

The arguments of the controller synthesis algorithm are constructed in the same way as in Section 6-1-3, employing the same values for `maxDepth` and `maxRefineDepth` (i.e., 30 and 10).

The synthesis of the controller for the current benchmark problem terminated after 215 iterations[15], at which point the coverage of the controller no longer increases with each iteration[16].

Figures 6-13 and 6-14 show heat maps of respectively the control action a and the value function V for each state in the range for which the controller is generated. Figure 6-14 also shows the phase plane of the closed-loop system. The blank regions in the plot indicate that no (allowed) control actions exist that can steer the state trajectory away from a constraint violation.

[14] This was done for inspection purposes of the generated controller; otherwise, using the range used in [32] would also be acceptable.

[15] These took around 664 minutes, using the same computer as was used in Section 6-1-3, i.e., a Macbook Pro with a 2.9 GHz Intel Core i7 processor.

[16] The generated controller covers about 64% of the states specified in (6-26) and (6-27).

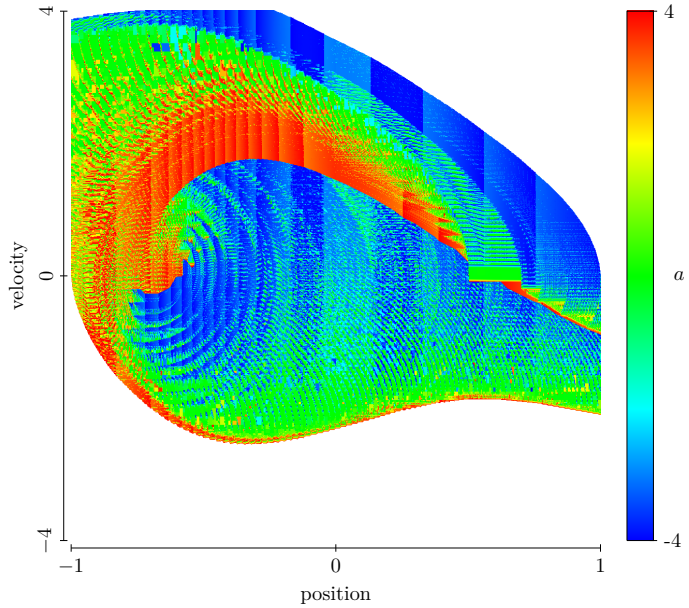


Figure 6-13: Heat map of the control actions.

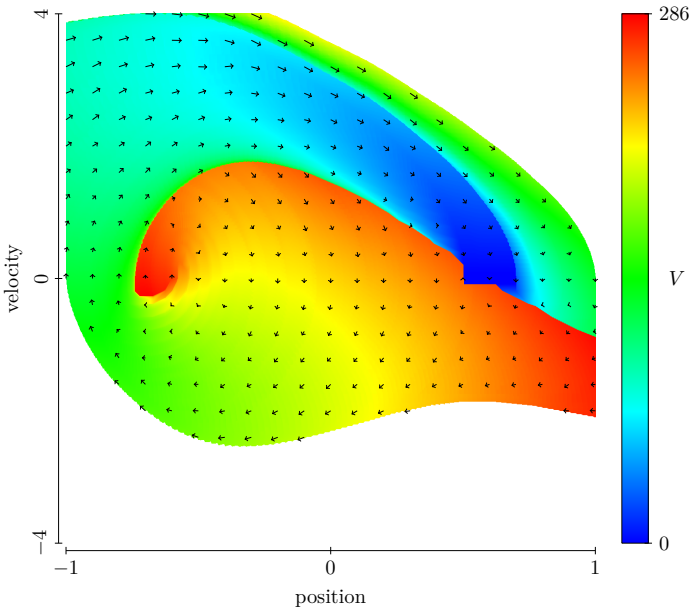


Figure 6-14: Phase plane of the closed-loop system and heat map of the value function V.

6-2-4 Closed-loop system

As in [32] the controller consists of two parts: one part that handles the control outside the specified target set, and one part that handles control inside the target set. The controller used outside the target set is generated using the presented controller synthesis algorithm, but unlike the benchmark problem of Section 6-1 the generated controller is not used when the target set is reached.

Inside the target set a *bang-bang controller* is used, as in [32], to keep the state inside the target set. This bang-bang controller consists of the saturated (between -4 and 4) sum of two large gains, K_p and K_v, that are respectively multiplied with the position and velocity errors, $x_\mathrm{m} - x$ and $\dot{x}_\mathrm{m} - \dot{x}$. The values of K_p and K_v have been determined using *trial and error*[17].

The parametrisation of the controller (5-3) can now be written as:

$$g\left(\begin{bmatrix} x \\ \dot{x} \end{bmatrix}, \Phi\left(x\right)\right) = \begin{cases} \min\Big(4,\ \max\left(-4,\ K_\mathrm{p}(x_\mathrm{m} - x) + K_\mathrm{v}(\dot{x}_\mathrm{m} - \dot{x})\right)\Big) & \text{if } x \in X_\mathrm{target} \\ \Phi\left(\begin{bmatrix} x \\ \dot{x} \end{bmatrix}\right) & \text{otherwise} \end{cases} \tag{6-31}$$

with $X_\mathrm{target} = \begin{bmatrix} x_\mathrm{target} \\ \dot{x}_\mathrm{target} \end{bmatrix}$ (as specified in (6-24) and (6-25)), $K_\mathrm{p} = 10^6$, $K_\mathrm{v} = 10^6$, $x_\mathrm{m} = 0.6$, and $\dot{x}_\mathrm{m} = 0.0$ (x_m and $\dot{x}_\mathrm{m}$ are set to the midpoints of respectively x_target and $\dot{x}_\mathrm{target}$).

Figure 6-15 shows a number of trajectories of the closed-loop system for different initial states, with small circles indicating the initial states and the square indicating the target set. Figure 6-16 shows the state and control actions for one of these trajectories.

In Figure 6-16 the difference between the behaviour of the two controllers is clearly visible, with the bang-bang controller taking over at around $t = 2$ seconds when the target set, indicated with the dashed lines, is reached.

The main reason for using a separate controller when the target set is reached is that not all states around the target can reach the target set within a reasonable number of time steps (unlike in Section 6-1 where the generated controller can steer the state back to the target set in one or two time steps). This can be clearly seen around the target set in Figure 6-14 where outside the lower bounds of the target set the values of V are around its maximum, while around the upper bounds of the target set the values of V are around its minimum.

[17] This approach was used due to the absence of a structured way of obtaining these values and the fact that we are only dealing with two parameters.

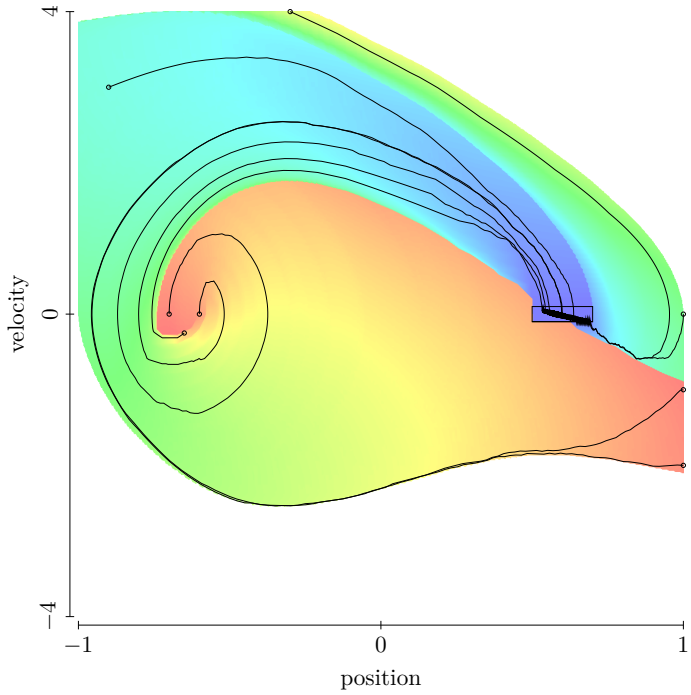


Figure 6-15: Example trajectories of the closed-loop system.

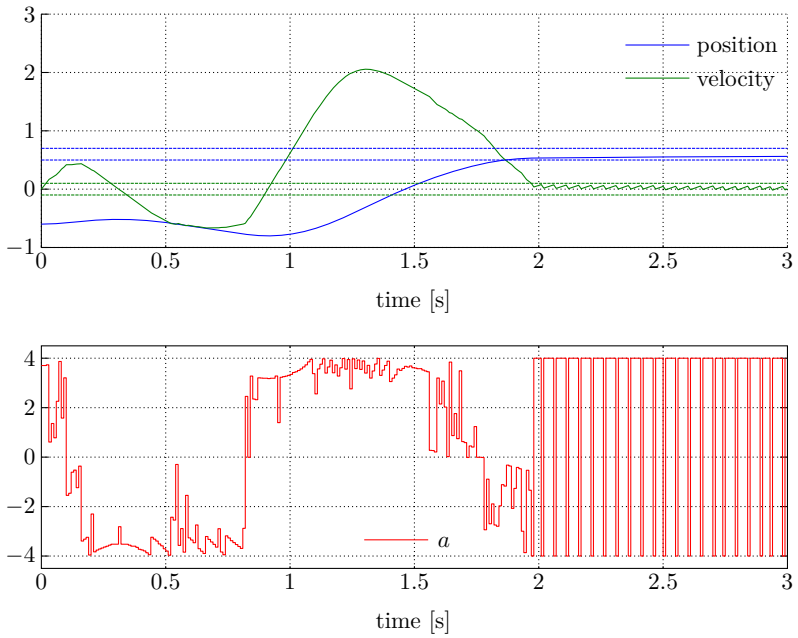


Figure 6-16: Closed-loop response and control action of the *car on the hill* benchmark for $x_0 = -0.6$ and $\dot{x}_0 = 0.0$. The dashed lines in the top sub-figure are used to indicate the target set.

6-3 Summary

In this chapter it has been shown how the controller synthesis algorithm presented in Chapter 5 can be used to generate controllers for the *pendulum swing-up and stabilisation* and *car on the hill* benchmark problems. The viability of the generated controllers have been tested by using them to control continuous-time models[18] of the plants.

In Section 6-1-3 it has been shown how the synthesis algorithm can be extended to support problems for which it was not originally created, as was demonstrated for the *pendulum swing-up and stabilisation* where the range of the angles was mapped to the interval $[0, 2\pi]$.

Moreover in Section 6-2-1 it was shown how a custom interval extension of the dynamics could be constructed that is superior to the natural interval extension of the dynamics.

[18] Approximated using the Dormand-Prince method.

Chapter 7

Conclusions and recommendations

This chapter summarises the main results of this thesis and presents (potential) improvements of the synthesis algorithm and topics for further research.

7-1 Conclusions

The main conclusion of this thesis is that it is possible to describe, implement, and successfully demonstrate a novel synthesis algorithm for the generation of state feedback controllers for *constrained non-linear discrete-time systems* (or for systems that can be described as such) based on *backwards induction* using techniques based on *interval methods*.

The described synthesis algorithm uses *interval methods* to implement *general-purpose* representations of closed sets and operations on these sets. By extending the closed-set representations with cost information and controller parameters it was made possible to represent the controller parameters and cost-to-go for entire regions of the state space and update them in an efficient way when new and/or improved values become available during following iterations of the synthesis algorithm.

In each iteration of the synthesis algorithm the set of *backward reachable* states of the previous set of backwards reachable states is determined together with their optimal controller parameters (for a pre-specified parametrised state-feedback controller) and associated cost-to-go and merged with the previously determined set of controller parameters and cost-to-go values. By repeating these iterations until a stopping condition is satisfied optimal controller parameters for all states within a pre-specified region of the state-space are obtained, thus generating an optimal controller.

The practical implementation of the described algorithm[1], made possible by the use of interval methods[2], allowed us to solve non-linear constrained derivative-free global optimisation problems and to determine backward reachable states while imposing only modest requirements on the models, cost function, and constraints used:

- the dynamics of the process to be controlled can be described by a difference equation or can be approximated by one,
- *interval extensions* of the expressions of the dynamics, cost function, and constraints can be constructed,
- the dimensionality of the state space and controller parametrisation is small enough to keep the computational and spatial complexity of the synthesis algorithm within acceptable bounds.

While these requirements limit the applicability of the presented synthesis algorithm, it was still able to successfully generate viable controllers for non-trivial benchmark problems, namely the *pendulum swing-up and stabilisation* and the *car on the hill* benchmark problems. The viability of the generated controllers was successfully demonstrated by using them to regulate continuous-time models of the benchmark problems.

It must be noted that the benchmark problems used in this thesis have been chosen to illustrate various aspects of the synthesis algorithm and involve only the generation of controllers for relatively simple mechanical systems. The synthesis algorithm can however be applied to a much wider range of problems only restricted by the limitations mentioned above. The requirement that the dynamics for which the controller is generated, is described by a difference equation is not a significant limitation since continuous-time processes can (in general) be approximated by a discrete-time process using a sufficiently small time step and an appropriate numerical integration method. The requirement that interval extensions can be constructed for the real-valued functions that describe the dynamics, cost function, and constraints, also limits the applicability of the synthesis algorithm, but since this requirement tends to be easily satisfied (by requiring that these functions can be expressed as *closed form expressions*[3]) the algorithm can still be used for a large range of controller synthesis problems.

If we take the limitations of the current implementation of the synthesis algorithm into account, and assuming the effects of the curse of dimensionality can be further mitigated (potential approaches for achieving this are presented in Sections 7-2 and 7-3), the synthesis algorithm can be used for generating controllers for problems in a wide range of fields.

[1] Using the C programming language [20].

[2] By employing the *C-XSC* class library [24].

[3] Being a *closed-form expression* is a *sufficient* condition for being able to construct an *interval extension* of an expression.

The following list gives examples of these fields and references to descriptions of specific control problems:

- Healthcare: management of diabetes [19], cancer treatment [14],
- Process industry: food processing [21], tank level control [29], bioreactor control [40],
- Automotive: throttle control [52], adaptive cruise control [38],
- Power systems: power conversion [28], control of fuel cell systems [42],

where the papers referenced with these items solve their control problems using various techniques, e.g., *sliding mode control*, *neural networks*, *hybrid methods*, or *explicit MPC*. These papers all have in common that they present control problems that can potentially be solved using the controller synthesis method presented in this thesis; they are relatively low dimensional, i.e., they have a relatively small number of states and inputs, but are (highly) non-linear, i.e., it is not feasible to create suitable controllers by using linearised plant models.

7-2 Recommendations

In the current section we will focus on techniques that potentially can reduce the effects of the curse of dimensionality.

The implementation of the synthesis algorithm given in Section 5-3 should be considered a very basic implementation of the algorithm described in Section 5-2. Although the current implementation is fully functional for real-valued problems and can be extended to mixed integer problems (through appropriate controller parametrisation), the efficiency of the implementation can be improved significantly (at the cost of a higher complexity) by various techniques described in Section 7-2-1.

In Section 7-2-2 it will be shown how specific properties of the dynamics of the system for which the controller is generated can be exploited to reduce the computational complexity of the controller synthesis.

Finally, a short discussion on how the general-purpose operations that are currently used by the synthesis algorithm can be optimised for specific use in the synthesis algorithm is given in Section 7-2-3.

7-2-1 Efficiency

In this section possible improvements to the synthesis algorithm and its implementation will be presented. These improvements do not extend the functionality of the synthesis algorithm but are purely aimed at increasing its efficiency, i.e. lowering its time and/or space complexity. It must be noted that for most of the suggested improvements a trade-off has to be made between time and space complexity, where we want to minimize the time complexity and have a constraint on the space complexity, i.e., make it as fast as possible while using a limited amount of memory.

Create parallel implementations of various algorithms

Since both the SIVIA and the Moore-Skelboe algorithms are almost[4] *embarrassingly parallel*[5] and the controller synthesis algorithm presented in this thesis is a combination and extension of these algorithms, the creation of a parallel version of the synthesis algorithm is something that should be considered.

Eliminate decimation

In the current implementation of the synthesis algorithm decimation (see item (3b) in Section 5-3-3) is used as a crude but effective way of limiting spatial complexity *during synthesis*, but it has as a side effect that potentially optimal (or good enough) controller parameters are discarded.

The generated controllers can be made more compact by not using decimation *during* controller synthesis and use the resulting increase of viable controller parameters for each leaf node of the binary tree representing the Regular Sub-Paving (RSP) of the controller to *potentially* merge sibling leaf nodes.

This approach is comparable to the `reunite` function described in Algorithm 5, that merges sibling nodes having the same set-theoretic value. The difference between the current approach compared and the `reunite` function is that we now try to merge *sets* of controller parameters rather than set-theoretic values. We can do this by using the fact that if the sets of controller parameters associated with the sibling nodes have a non-empty intersection the values in this intersection are valid for *both* sibling nodes, which than can be merged, i.e., the nodes can be removed and their parent node becomes a leaf node with the intersection of the sets of controller parameters of its original child nodes as its controller parameters. As with the `reunite` function this approach can be used recursively until all existing, and newly generated leaf nodes that can be merged, are merged.

Eliminate redundancy

In the current implementation of the synthesis algorithm each node of the binary tree representing an RSP contains an entry for the value of the box associated with the node. The advantage of this is that the value of the box is directly accessible, and calculation of the box values is straightforward (see Algorithm 8). The disadvantages of this approach are that all these boxes are stored, even when they are no longer used, the boxes have to be reconstructed each time an RSP is generated, and that duplicate midpoint calculations occur.

One way of avoiding the duplication of midpoints is to store them in a separate binary tree for each axis that contains all the midpoints required in a worst-case scenario, i.e., when the binary tree representing the RSP obtained during set inversion is a *perfect* binary tree[6]. The

[4] Their computational graphs are both binary trees. By carrying out a centralised breadth-first traversal (up to a specified recursion depth) followed by parallel depth-first traversals of the resulting leaf-nodes we can get a speed up that is almost proportional to the number of tasks that can be run in parallel.

[5] *Embarrassingly parallel* is a commonly used term, see for example Section 2.4.2 of [41].

[6] A *perfect* binary tree [4] is a binary tree where all leaf nodes have the same depth and each parent node has exactly two child nodes.

binary tree containing the midpoints for each axis is in this case also a *perfect* binary tree. This has as an advantage the binary tree can be represented as an array without any overhead.

Restriction of optimised states

Another improvement that should be considered is to limit the set of states for which the optimal controller parameters are to be determined to a limited number of previous frontiers plus the current frontier. One should think of the frontier as the set of states that in the current iteration can reach the set of *target* states but were not able to do so in the previous iterations. We define the frontier X_{f} and the set of states $X_{\mathrm{f},n,i}$ for which optimal controller parameters are to be determined in iteration i as:

$$X_{\mathrm{f},i} = X_i \cap \left(X^*_{i+1}\right)^{\mathrm{c}} \tag{7-1}$$

$$X_{\mathrm{f},n,i} = \bigcup_{j=i}^{i+n} X_{\mathrm{f},j} \tag{7-2}$$

where n specifies the number of previous frontiers that are used in the current iteration and $\left(X^*_{i+1}\right)^{\mathrm{c}}$ is the *complement* of X^*_{i+1}.

The rationale behind this approach is the observation that the likelihood that an improved controller parameter for a state (once a state is backwards reachable) can be found, decreases with each iteration of the synthesis algorithm.

The optimal choice of the number of previous frontiers n (where $n \geq 0$) depends on the system dynamics, cost function used, and imposed constraints but is likely to be relatively low[7]. Moreover while the optimal value of n can be easily observed retrospectively by inspection of the evolution of the controller that is generated, the question of how to obtain an a priori optimal value of n is a topic that still needs to be studied.

7-2-2 Problem specific optimisations

In this section a number of *non general-purpose* techniques are discussed that can improve the efficiency of the synthesis algorithm. These techniques exploit specific properties that the system, for which the controller is to be generated, might have.

Exploit symmetry

When the resulting controller g and cost-to-go V exhibit symmetry, the space and time complexity of the controller synthesis can be potentially reduced. For example when $g(x_{\mathrm{s}}+x) = -g(x_{\mathrm{s}}-x)$ and $V(x_{\mathrm{s}}+x) = V(x_{\mathrm{s}}-x)$, as in the case of the pendulum benchmark problem (see Section 6-1) (where $x_{\mathrm{s}} = \left[\begin{smallmatrix}\pi\\0\end{smallmatrix}\right]$) it is sufficient to calculate only the controller for $0 \leq \theta \leq \pi$ and $-6 \leq \dot{\theta} \leq 6$ and 'virtually' extend these results to cover the range of $\pi \leq \theta \leq 2\pi$.

[7] Inspection of the evolution of the controller parameters in the *pendulum swing-up and stabilisation* benchmark problems revealed that, depending on the cost function used, a value of around 5 or even 0 could be used for n without affecting the optimality of the generated controller.

Reduce duplicate evaluations

The efficiency of the implementation of the synthesis algorithm can be improved if the interval extension of the dynamics of the system, $[f_{\mathrm{g}}]$, can be decomposed as:

$$[f_{\mathrm{g}}](x,\phi) = [f_{\mathrm{gx}}](x) + [f_{\mathrm{gp}}](\phi) \tag{7-3}$$

If this decomposition can be made the results of the evaluation of $[f_{\mathrm{gp}}](\phi)$ and $[f_{\mathrm{gx}}](x)$ can be re-used after bisection of x and ϕ, respectively. While this method is a clear improvement it causes a loss of generality, e.g., it could be used with the pendulum benchmark problem (Section 6-1), but not with the car on the hill benchmark problem (Section 6-2) due to the occurrence of both a state x and a control action a in xdd_1. If the dynamics cannot be decomposed like (7-3) a partial decomposition could also be considered:

$$[f_{\mathrm{g}}](x,\phi) = [f_{\mathrm{gx}}](x) + [f_{\mathrm{gpx}}](x,\phi) \tag{7-4}$$

This partitioning can be used with (6-21) where $[f_{\mathrm{gx}}]$ and $[f_{\mathrm{gpx}}]$ would correspond to xdd_2 and xdd_1 respectively.

Explicit support for (mixed-)integer problems

In this paragraph we show how the synthesis algorithm can be extended to support integer and mixed-integer controller parametrisations in an efficient[8] way.

Synthesis of controllers that can be parametrised using a finite set of integers, i.e., $\Phi = \{1, 2, \ldots, n\}$, can be implemented by determining the backwards reachable set of states independently for each $i \in \Phi$ and taking the union of the resulting backwards reachable sets:

$$X_k = \bigcup_{i \in \Phi} \left(f_{\mathrm{g},i}^{-1}(X_{k+1}) \cap X_{\mathrm{cs},i} \right) \tag{7-5}$$

with

$$f_{\mathrm{g},i}(x) = f_{\mathrm{g}}(x,i) \tag{7-6}$$

$$X_{\mathrm{cs},i} = h_{\mathrm{g},i}^{-1}((-\infty, 0]) \tag{7-7}$$

$$h_{\mathrm{g},i}(x) = h_{\mathrm{g}}(x,i) \tag{7-8}$$

For integer controller parametrisations the practical implementation of the optimisation step (5-27) becomes simpler since bisection of the controller parameters is no longer required, only bisection of the states is still required, i.e., the bisection step in the `optimizeLists` function shown in Algorithm 28 can be omitted.

[8] It must be noted that the current implementation can support integer and mixed-integer controller parameters with some small modifications to the bisection operator and stopping criteria (since $\mathbb{Z} \subset \mathbb{R}$), but doing so would likely be less efficient than the approaches presented in this section.

To support mixed-mixed integer controller parameters, i.e., $\Phi_{\mathbb{R}} \times \Phi_{\mathbb{Z}} \subset \mathbb{R}^n \times \mathbb{Z}$, we can use an approach similar to that used with the integer controller parametrisation but with a projection step added:

$$X_k = \bigcup_{i \in \Phi_{\mathbb{Z}}} \pi_{\mathrm{x}}\Big(f_{\mathrm{g},i}^{-1}(X_{k+1}) \cap (X_{\mathrm{cs},i}, \Phi_{\mathrm{cs},i}) \Big) \tag{7-9}$$

with

$$f_{\mathrm{g},i}(x, \phi_{\mathbb{R}}) = f_{\mathrm{g}}(x, \left[\begin{smallmatrix} i \\ \phi_{\mathbb{R}} \end{smallmatrix}\right]) \tag{7-10}$$

$$(X_{\mathrm{cs},i}, \Phi_{\mathrm{cs},i}) = h_{\mathrm{g},i}^{-1}\big((-\infty, 0]\big) \tag{7-11}$$

$$h_{\mathrm{g},i}(x, \phi_{\mathbb{R}}) = h_{\mathrm{g}}(x, \left[\begin{smallmatrix} i \\ \phi_{\mathbb{R}} \end{smallmatrix}\right]) \tag{7-12}$$

where $\phi_{\mathbb{R}}$ is the real-valued valued component of the parameter vector ϕ and $\Phi_{\mathbb{Z}}$ is the set of all integers used in the parametrisation.

When using mixed-integer controller parameters the optimisation step (5-27) still requires bisection of the controller parameters, although bisection of the integer components can be omitted.

Equation (7-5) or (7-9) would replace (5-25) and (5-26) in the controller synthesis algorithm described in Section 5-2.

7-2-3 Application specific operator implementations

The operations used in the current implementation of the controller synthesis algorithm were constructed by extending the implementation of the *set computation* operations described in Chapter 4. While this is a perfectly valid approach, it certainly leaves room for improvement. This is due to the fact that the set computation operations described in Chapter 4 are implemented as operations without *side effects*, e.g., when evaluating $C \longleftarrow A \cup B$ the binary trees that describe the RSPs representing the sets A and B are unchanged after evaluation of the union. When it is known beforehand that one or more of the argument sets of the union operator are no longer used after the union has been determined, we can obtain a significant reduction in the number of memory operations (allocation, de-allocation, and duplication) by moving the *pointers* to their root nodes around rather than duplicating sub-trees. This approach will modify the RSPs representing A and/or B, which is not an issue if they are no longer used.

7-3 Topics for future research

The following paragraphs introduce a number of topics that can potentially improve efficiency or applicability of the synthesis algorithm, but that at this time have not been sufficiently researched.

The interval arithmetic operations and interval extensions used in this thesis do not retain the dependency between the arguments and outputs of a function evaluation (see Section 2-3-1). This together with the *wrapping effect* (see Section 2-3-2) are the main causes of the overestimation in interval calculations. By using higher-order methods like *Affine Arithmetic* [7, 49], which aims to retain first-order relations between the input argument(s) and the results of operations we can reduce both the effects of the loss of dependency and the wrapping effect. The wrapping effect is reduced since the joint range of two or more affine forms is a *zonotope*[9] rather than a *box* as is the case with two or more intervals. Methods that (aim to) retain even higher-order relations also exist [31] [30], but their implementation and evaluation is significantly more complex than that of affine arithmetic while the expected efficiency gains obtained by their use are unlikely to be as significant as those obtained by using affine arithmetic rather than interval arithmetic.

Obtaining computational and spatial complexity measures that are representative for practical usage scenarios of the synthesis algorithm is a topic that has not been investigated. The computational and spatial complexity of the various algorithms used in the synthesis method is $\mathrm{O}(2^n)$. But the complexity that is observed when running the synthesis algorithm for the benchmark problems is noticeably lower. There are a number of methods for obtaining (approximate) complexity characterisations for practical cases like *smoothed analysis* [48] and it might be worthwhile to apply them to the various algorithms used in the synthesis method.

The use of *Bernstein polynomials* [26] for creating improved interval extensions is also a topic worth investigating further. A number of different papers have used Bernstein polynomials (some in combination with Taylor polynomials) as an alternative to conventional interval extensions and reported improved performance of the resulting algorithms [10, 13, 37, 1].

Binary trees (representing regular sub-pavings) are the basic data structure used in the synthesis algorithm. These binary trees together with the data stored within their nodes is responsible for the bulk of the memory used by the synthesis algorithm. Currently these binary trees are implemented as nodes with pointers to their child nodes. The advantage of this representation is that it is straightforward to implement and use but it has as disadvantage that its spatial complexity can be prohibitive. The spatial complexity of the binary trees can be reduced by using more succinct representations (see for example [36]) at the cost of an increased complexity of the data structures and computational complexity of their use.

[9] A *zonotope* is a centre-symmetric convex polytope, i.e., a convex polytope that is symmetric around some point.

Bibliography

[1] M. Arounassalame, "Analysis of nonlinear electrical circuits using bernstein polynomials," *International Journal of Automation and Computing*, vol. 9, no. 1, pp. 81–86, 2012. [Online]. Available: http://dx.doi.org/10.1007/s11633-012-0619-3

[2] R. Bellman, *Dynamic Programming*. Dover Publications, 1957.

[3] J. Black, N. Hashimzade, and G. Myles, *A Dictionary of Economics*. Oxford University Press, Oxford, 2009.

[4] P. E. Black. (2004) Dictionary of Algorithms and data Structures. National Institute of Standards and Technology, U.S. Departement of Commerce. [Online]. Available: http://xlinux.nist.gov/dads/HTML/perfectBinaryTree.html

[5] J. M. Borwein and R. E. Crandall, "Closed forms: what they are and why we care," Notices of the American Mathematical Society, Volume 60, Number 1, 2010.

[6] F. E. Cellier and E. Kofman, *Continuous System Simulation*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[7] J. L. D. Comba and J. Stolfi, "Affine arithmetic and its applications to computer graphics," in *Proceedings of VI SIBGRAPI 1993.*, 1993, pp. 9–18, recife, Brazil, October 1993.

[8] A. E. Csallner, T. Csendes, and M. Csaba Markót, "Multisection in interval branch-and-bound methods for global optimization - I. theoretical results," *Journal of Global Optimization*, vol. 16, no. 4, pp. 371–392, Apr. 2000. [Online]. Available: http://dx.doi.org/10.1023/A:1008354711345

[9] T. Csendes, "New subinterval selection criteria for interval global optimization," *Journal of Global Optimization*, vol. 19, no. 3, pp. 307–327, Mar. 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1011228208051

[10] T. Dang and R. Testylier, "Reachability analysis for polynomial dynamical systems ssing the Bernstein expansion," *Reliable Computing*, vol. 17, no. 2, pp. 128–152, 2012.

[11] M. Daumas, G. Melquiond, and C. Muñoz, "Guaranteed proofs using interval arithmetic," in *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, P. Montuschi and E. Schwarz, Eds., Cape Cod, MA, USA, 2005, pp. 188–195.

[12] J. J. Dennis and R. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, ser. Prentice-Hall series in computational mathematics. Prentice-Hall, 1983, no. 16, iSBN: 0-13-627216-9. [Online]. Available: http://discover.tudelft.nl:8888/recordview/view?recordId=aleph%3A000291957

[13] S. Foufou and D. Michelucci, "The Bernstein basis and its applications in solving geometric constraint systems," *Reliable Computing*, vol. 17, no. 2, pp. 192–208, 2012.

[14] M. Hadjiandreou and G. Mitsis, "Mathematical modeling of tumor growth, drug-resistance, toxicity, and optimal therapy design," *Biomedical Engineering, IEEE Transactions on*, vol. 61, no. 2, pp. 415–425, Feb 2014.

[15] IEEE, "IEEE Standard for Floating-Point Arithmetic," Microprocessor Standards Committee of the IEEE Computer Society, New York, NY, USA, Tech. Rep., Aug. 2008. [Online]. Available: http://dx.doi.org/10.1109/IEEESTD.2008.4610935

[16] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter, *Applied Interval Analysis, with Examples in Parameter and State Estimation, Robust Control and Robotics*. London: Springer-Verlag, 2001.

[17] L. Jaulin and E. Walter, "Guaranteed nonlinear parameter estimation from bounded-error data via interval analysis," *Mathematics and Computers in Simulation*, vol. 35, no. 2, pp. 123 – 137, 1993. [Online]. Available: http://www.sciencedirect.com/science/article/pii/037847549390008I

[18] ——, "Set inversion via interval analysis for nonlinear bounded-error estimation," *Automatica*, vol. 29, no. 4, pp. 1053 – 1064, 1993. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0005109893901064

[19] P. Kaveh and Y. Shtessel, "Blood glucose regulation via double loop higher order sliding mode control and multiple sampling rate," in *Modern Sliding Mode Control Theory*, ser. Lecture Notes in Control and Information Sciences, G. Bartolini, L. Fridman, A. Pisano, and E. Usai, Eds. Springer Berlin Heidelberg, 2008, vol. 375, pp. 427–445. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-79016-7_20

[20] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Prentice Hall Professional Technical Reference, 1988.

[21] M. Khadir and J. Ringwood, "Linear and nonlinear model predictive control design for a milk pasteurisation plant," in *Control and Intelligent Systems*, vol. 31, no. 1, 2003.

[22] M. Kieffer, L. Jaulin, I. Braems, and E. Walter, "Guaranteed set computation with subpavings." *Scientific Computing, Validated Numerics, Interval Methods*, pp. 167–178, 2001. [Online]. Available: http://www.ensta-bretagne.fr/e3i2/images/E3I2/Publi/publi/KIEFFER_01a.pdf

[23] S. C. Kleene, "On notation for ordinal numbers," *Journal of Symbolic Logic*, vol. 3, no. 4, pp. 150–155, 1938.

[24] W. Kramer, "C-XSC: A powerful environment for reliable computations in the natural and engineering sciences," in *Biomedical Engineering and Informatics (BMEI), 2011 4th International Conference on*, vol. 4, 2011, pp. 2130–2134.

[25] R. Krawczyk and A. Neumaier, "Interval slopes for rational functions and associated centered forms," *SIAM Journal on Numerical Analysis*, vol. 22, no. 3, pp. pp. 604–616, 1985. [Online]. Available: http://www.jstor.org/stable/2157086

[26] G. G. Lorentz, *Bernstein Polynomials*, 2nd ed. AMS Chelsea Publishing, 1986, vol. 323. [Online]. Available: http://www.ams.org/bookstore-getitem/item=CHEL/323.H

[27] V. P. Madan, "Interval Newton method: Hansen-Greenberg approach-some procedural improvements," *Applied Mathematics and Computation*, vol. 35, no. 3, pp. 263 – 276, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0096300390900466

[28] A. A. A. Mahmood Mirzaei, "Hybrid modeling of the non-inverting buck-boost dc-dc converter," in *International Conference on Control, Automation and Systems*, Seoul, Korea, 2008.

[29] J. Malmborg and J. Eker, "Hybrid control of a double tank system," in *IEEE Conf. Control Applications*, 1997, pp. 137–138.

[30] F. Messine, "Extentions of affine arithmetic: Application to unconstrained global optimization," *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 992–1015, nov 2002. [Online]. Available: http://www.jucs.org/jucs_8_11/extentions_of_affine_arithmetic

[31] F. Messine and A. Touhami, "A general reliable quadratic form: An extension of affine arithmetic," *Reliable Computing*, vol. 12, pp. 171–192, 2006, 10.1007/s11155-006-7217-4. [Online]. Available: http://dx.doi.org/10.1007/s11155-006-7217-4

[32] A. Moore, "Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces," in *Machine Learning: Proceedings of the Eighth International Conference*, L. Birnbaum and G. Collins, Eds. San Francisco, CA, USA: Morgan Kaufmann, June 1991.

[33] R. E. Moore, "Global optimization to prescribed accuracy," *Computers & Mathematics with Applications*, vol. 21, no. 6 - 7, pp. 25 – 39, 1991. [Online]. Available: http://www.sciencedirect.com/science/article/pii/089812219190158Z

[34] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2009.

[35] R. Munos and A. Moore, "Variable resolution discretizations for high-accuracy solutions of optimal control problems," in *Proceedings of the International Joint Conference on Artificial Intelligence, Stockholm*, 1999, pp. 1348–1355.

[36] J. I. Munro, V. Raman, and A. J. Storm, "Representing dynamic binary trees succinctly," in *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '01. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2001, pp. 529–536. [Online]. Available: http://dl.acm.org/citation.cfm?id=365411.365526

[37] P. S. V. Nataraj and K. Kotecha, "Global optimization with higher order inclusion function forms. part 1: A combined Taylor-Bernstein form," *Reliable Computing*, vol. 10, no. 1, pp. 27–44, 2004.

[38] G. Naus, J. Ploeg, M. V. de Molengraft, W. Heemels, and M. Steinbuch, "Design and implementation of parameterized adaptive cruise control: An explicit model predictive control approach," *Control Engineering Practice*, vol. 18, no. 8, pp. 882 – 892, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0967066110000882

[39] A. Neumaier, *Interval Methods for Systems of Equations*, ser. Encyclopedia of Mathematics and Its Applications 37. Cambridge University Press, 1990, iSBN: 0-521-33196-X.

[40] M. Onder Efe, "Model reference sliding mode control for bioreactor benchmark problem," in *Variable Structure Systems, 2008. VSS '08. International Workshop on*, 2008, pp. 361–366.

[41] P. Pacheco, *An Introduction to Parallel Programming*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[42] C. Panos, K. Kouramas, M. Georgiadis, and E. Pistikopoulos, "Modelling and explicit model predictive control for PEM fuel cell systems," *Chemical Engineering Science*, vol. 67, no. 1, pp. 15 – 25, 2012, dynamics, Control and Optimization of Energy Systems. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0009250911004842

[43] H. Ratschek and J. Rokne, "Interval global optimization," in *Encyclopedia of Optimization*, C. A. Floudas and P. M. Pardalos, Eds. Springer US, 2009, pp. 1739–1757. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-74759-0_306

[44] J. G. Rokne, *Interval Arithmetic and Interval Analysis: An introduction.* Heidelberg, Germany, Germany: Physica-Verlag GmbH, 2001, pp. 1–22. [Online]. Available: http://pages.cpsc.ucalgary.ca/~rokne/CPSC491/interval_art.pdf

[45] S. Rump, "INTLAB - INTerval LABoratory," in *Developments in Reliable Computing*, T. Csendes, Ed. Dordrecht: Kluwer Academic Publishers, 1999, pp. 77–104, http://www.ti3.tuhh.de/rump/.

[46] R. Sainudiin, "Machine Interval Experiments," Ph.D. dissertation, Cornell University, 2005. [Online]. Available: http://www.math.canterbury.ac.nz/~r.sainudiin/preprints/PHD.pdf

[47] S. Skelboe, "Computation of rational interval functions," *BIT*, vol. 14, pp. 87 – 95, 1974. [Online]. Available: http://www.diku.dk/hjemmesider/ansatte/stig/CompRatIntv.pdf

[48] D. Spielman and S.-H. Teng, "Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time," in *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, ser. STOC '01. New York, NY, USA: ACM, 2001, pp. 296–305. [Online]. Available: http://doi.acm.org/10.1145/380752.380813

[49] J. Stolfi, L. de Figueiredo, and C. P, "An introduction to affine arithmetic," 2003.

[50] J. M. van Ast, "Ant Colony Optimization for Control," Ph.D. dissertation, TU Delft, 2010.

[51] M. van Emden and B. Moa, *Frontiers in Global Optimization*, ser. Nonconvex Global Optimization and Its Applications. Kluwer Academic Publishers, 2003, ch. Termination criteria in the Moore-Skelboe Algorithm for Global Optimization by Interval Arithmetic. [Online]. Available: http://webhome.cs.uvic.ca/~vanemden/Publications/mooreSkelb.pdf

[52] F. P. Y. Vidal, L. Acho, "Robust control of an electronic throttle system via switched chattering control: benchmark experiments," in *2009 IFAC Workshop on Engine and Powertrain Control, Simulation and Modeling, E-CoSM '09*, Paris, France, 2009.

[53] A. Zomorodian and H. Edelsbrunner, "Fast software for box intersection," *International Journal of Computational Geometry & Applications*, vol. 12, pp. 143–172, 2002.

Glossary

List of Acronyms

IA	Interval Arithmetic
RSP	Regular Sub-Paving
SIVIA	Set Inversion Via Interval Analysis

List of Symbols

$(X, \Phi)_k$	Joint range of the states and controller parameters that can reach the target states in $N - k$ time steps
$[a, b]$	Closed real-valued interval: $\{x \in \mathbb{R} \mid a \leq x \leq b\}$
$[f]$	Interval extension of the real-valued function f
$[f]^*$	Interval-range of the real-valued function f
$[V]$	Interval-valued approximation of V
$\mathbb{IR}$	Set of real-valued *intervals*
$\mathbb{IR}^*$	Set of *extended-intervals*
$\mathbb{R}$	Set of *reals*
$\mathbb{Z}$	Set of *integers*
$\mathcal{RSP}(X)$	Set of all RSPs of box X
$\overline{X}$	Upper bound of interval X
ϕ	Parameter vector
$\Phi(x)$	Parameter function, returns the parameter vector associated with state x
Φ_k^*	Controller parameters associated with X_k^*
$\pi_{\mathrm{x}}(Z)$	Projection of Z onto the state-space
$\pi_{\Phi}(Z)$	Projection of Z onto the parameter-space
$\sqcup$	Interval union operator

$\underline{X}$	Lower bound of interval X
A^{c}	Complement of set A
f	A function (typically the system dynamics)
$f^{(i)}(x)$	The ith derivative of function f with respect to x
f_{g}	Parametrised system dynamics
g	Parametrised controller
h	Inequality constraints
h_{g}	Parametrised inequality constraints
J	Cost function
J_{ug}	Parametrised control action dependent part of J
J_{u}	Control action dependent part of J
J_{x}	State dependent part of J
$m(X)$	Midpoint of interval X: $m(X) = \frac{1}{2}(\underline{X} + \overline{X})$
N	Time step at which the set of target states is reached
$V(x)$	Cost-to-go for state x
V_k^*	Cost-to-go associated with X_k^*
$w(X)$	Width of interval X: $w(X) = \overline{X} - \underline{X}$
X_k^*	Coverage of the state space by the controller at time step k
X^{c}	The complement of set X
X_k	Set of states that can reach the set of target states in $N - k$ time steps
`NaN`	Not a Number