Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

The present work was submitted to the LuFG Theory of Hybrid Systems

BACHELOR OF SCIENCE THESIS

EFFICIENT CONVERSION OF GEOMETRIC STATE SET REPRESENTATIONS FOR HYBRID SYSTEMS

EFFIZIENTE KONVERTIERUNG GEOMETRISCHER ZUSTANDSRAUM-DARSTELLUNGEN FÜR HYBRIDE SYSTEME

Simon Froitzheim

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Jürgen Giesl

Additional Advisor:
Stefan Schupp, M.Sc.

Aachen, May 18, 2016

Abstract

Reachability analysis of linear hybrid systems via flow pipe computation makes extensive use of geometric and symbolic state set representations of various kinds. During analysis, several operations on state sets are performed and the complexity of these operations highly depends on the utilised representation.

With the transformation into a currently more advantageous representation being a valid approach, so as to improve runtime and precision, developing efficient conversions between different types of state set representations is advisable.

This thesis presents various procedures concerning the efficient conversion between convex polytopes (H- and V-representation), hyperrectangles, zonotopes and support functions.

Conducted evaluations show that it is difficult to provide efficient approaches and implementations for every possible conversion between the mentioned state set representations, albeit most of the introduced algorithms work both correctly and efficiently.

Eidesstattliche Versicherung

Name, Vorname

Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift

Acknowledgements

While this bachelor thesis and my study as a whole has been a difficult journey for me, especially in terms of my personal health, I still learned a lot during this phase of my life, not only about computer science, but also about what matters in life. It is safe to say that, without the support and motivation of family and friends, I would not have been able to achieve what I achieved by now.

Very special thanks thus go to my parents, Carina, Lars and Felix for their great support. Moreover, I would like to thank everyone else that encouraged me not only during this final challenge, but also during the entirety of my bachelor study, comprising every person that provided comfortable distractions from my work, also including my violin teacher Kalliopi.

Additionally, I am grateful to the whole staff of the i2 for providing a pleasant working atmosphere. Extra acknowledgments go to Prof. Dr. Erika Ábrahám for giving me the opportunity of writing this bachelor thesis and Prof. Dr. Jürgen Giesl for filling the role of my second examiner.

Last but not least, I would like to express my gratitude to my advisor Stefan Schupp, who was always open for questions and spent a lot of time helping me in the course of this thesis.

Contents

1 Introduction **9**

2 A Lead-In to Hybrid Systems **13**
2.1 Hybrid Automata 14
2.2 Reachability Analysis for Hybrid Systems 18
2.3 Controller Synthesis for Hybrid Systems 22

3 State Set Representations **23**
3.1 Convex Polyhedra 24
3.2 Hyperrectangles 26
3.3 Zonotopes 27
3.4 Support Functions 28
3.5 Other State Set Representations 30
3.6 Operations on State Sets 31

4 Conversion Procedures **35**
4.1 Principal Component Analysis and ORHs 36
4.2 Conversion of V-Polytopes 39
4.3 Conversion of H-Polytopes 44
4.4 Conversion of Hyperrectangles 46
4.5 Conversion of Zonotopes 49
4.6 Conversion of Support Functions 51

5 Evaluation **59**
5.1 General Analysis 59
5.2 Specific Experiments and Plots 65

6 Conclusion **69**
6.1 Summary 69
6.2 Discussion 69
6.3 Future work 70

Bibliography **71**

Chapter 1

Introduction

With the high complexity of today's technical systems and no decrease of this intricacy in sight, the verification of useful properties, predominantly safety, has never before been a more relevant task. This circumstance applies to the research field of hybrid systems as well, in fact, formal verification has proven itself to be somewhat problematic for hybrid systems: Many powerful verification tools for discrete automata already exist, but the same cannot be said about their discrete-continuous counterparts; the combination of discrete jumps and continuous flow that designates hybrid automata is still difficult to analyse.

Nonetheless various rudiments for verification purposes do exist: In most practical scenarios, simulation approaches still form the default verfication procedures, ranging from statistical methods that compute an amount of randomly chosen trajectories (e.g [CDL09]) to solutions that generate a finite set of trajectories through the system in a methodical manner (e.g. [DM07]). The largest problem with the idea of a simulation is that no matter how many simulation runs you conduct, there is always the possibility that you do not pass an unwanted but reachable state.

This very issue ensured the need to explore other concepts, with maybe the most important of these concepts being the reachability analysis, aiming to compute the set of *all* possible states in a given system. In our case, for hybrid systems, this very state set is often impossible to compute, hence it has to be approximated in the general case: Involving essentially three main categories for hybrid systems reachability analysis, consisting of theorem-proving-based, interval-constraint-propagation-based and fixed-point-computation-based approaches, I focus on the latter class that relies mostly on geometric objects for the approximation process.[1] It is thus no surprise that in context of this category, the possibilities of geometric as well as symbolic state set representations have been and are still being explored, resulting in a number of different methods often using distinct representations. Exemplary approaches feature the utilisation of linear dynamics [LG09, DB], predicate abstraction [ADI02] and level set functions [MBT01].

It soon became clear that it is tricky if not impossible to find an optimal state set representation, mainly because computational complexity highly diverges concerning each of the necessary operations for reachability analysis with varied representations: As an example, computing the union of two convex polyhedra in vertex representation may be easily done, but calculating the intersection is tough; using a half-

[1] For a more detailed look at these three classes, please refer to Section 2.2.

space representation for the polyhedron instead yields the opposite scenario in which the intersection operation is unproblematic with the union operation however being intricate to perform. Both operations are unfortunately important for fixed-point-computation-based reachability analysis, which proves to be a grave problem.

This state of affairs and the fact that some approaches like [ASB09] use multiple state set representations for their own concept of reachability analysis both give revelance to conversion procedures for state set representations: With transformations into other representations during the whole computation being a valid design concept, developing well-conceived procedures to do so can significantly improve the efficiency of the entire process. This means that not only the computation time benefits from sophisticated algorithms, but the mere precision of the conversion process likewise, since, as a matter of fact, exact conversions are rarely possible between the different state set representations because of their special properties, consequentially resulting in the need to approximate the solution. Even if the conversion could be done exactly, it is in many cases not advisable to do so, as an exact result requires a longer computation time than an approximation in the general case.

Existing research on the general topic of conversion of state set representations for hybrid systems reachability analysis can usually only be found in parts of specific reachability analysis approaches or in papers dealing with intrinsic state set representations: By way of example, specific reachability analysis approaches explore conversion partly in context of zonotope/hyperplane intersection [GLG08], zonotope/polytope intersection[ASB08], oriented rectangular hulls [SK03] and convex set overapproximation via support functions [LGG10, Var00].[1] On the other hand, paradigmatic essays with focus on state set representation feature representation-specific works (mostly zonotopes) that also deal with conversion like [Fuk04], [GNZ03] and [Fuk].

The conversion of state set representations plays an important role regarding controller synthesis for hybrid systems too, mainly in terms of underapproximative conversion [ABD+00].

There are two main objectives of this bachelor thesis: First and foremost the endeavour consists of developing and implementing various conversion algorithms under supervision as part of the *Toolbox for the Reachability Analysis of Hybrid Systems Using Geometric Approximations* (HyPro)[2] and secondly this final paper aims at presenting and evaluating those same algorithms in an accessible yet complete and both formal and substantial correct manner.

HyPro is in short a current project of the *Theory of Hybrid Systems* research group which is embedded into the *Software Modeling and Verification* chair at the RWTH Aachen University;[3] it consists of a C++ library that strives to give the possibility to evaluate and compare existing reachability analysis approaches regarding hybrid systems as well as to provide the means for the fast implementation of new techniques.

My bachelor thesis is structured as follows: Initially I give a more in-depth introduction to hybrid systems reachability analysis as well as describe a popular way of formally representing hybrid systems, namely hybrid automata, in Chapter 2. Sub-

[1] The possibilities of overapproximation via support functions and overapproximation utilising oriented rectangular hulls are explored more thoroughly in Chapter 4.

[2] HyPro can be found with `https://ths.rwth-aachen.de/research/projects/hypro/` (lastly called up by the 4th April, 2016)

[3] The homepage of the Software Modeling and Verification chair can be reached with the following hyperlink: `https://moves.rwth-aachen.de/` (lastly called up by the 4th April, 2016)

sequently, an overview over the most common geometrical and symbolic state set representations is presented in Chapter 3, before reaching the core of this thesis with the following two chapters that deal with the conversion of state set representations in specific: Chapter 4 describes the underlying conversion algorithms in detail, while Chapter 5 depicts an elaborate evaluation of these algorithms. A conclusion given with Chapter 6 summarises the results and discusses the success of the bachelor thesis and possible future work.

Chapter 2

A Lead-In to Hybrid Systems

Discrete systems are systems with discrete state changes in a possibly infinite state space. Examples for such systems are a computer program or a sensor in a water boiler which reports when the temperature of the water inside is above a certain threshold.

Dynamic systems are systems that behave continuously with a real-valued state space, ergo analog systems that measure physical quantities like time, temperature or speed.

A *hybrid system* combines discrete and continuous behaviour (cf. Figure 2.1) and can basically be everything from a simple ball that is thrown and bounces off the ground up to a complex airplane: An exemplary "bouncing ball" has its current velocity and height as continuous components while the time points when the ball bounces off the ground introduce discrete events which gives the contingency to model it as a hybrid system. Modern airplanes run with complex software which is fully discrete, but for all that the software operates in an environment that demands for the application of a whole lot of physical systems so as to deal with many continuous factors like temperature, air pressure, and in this case yet again height and velocity.

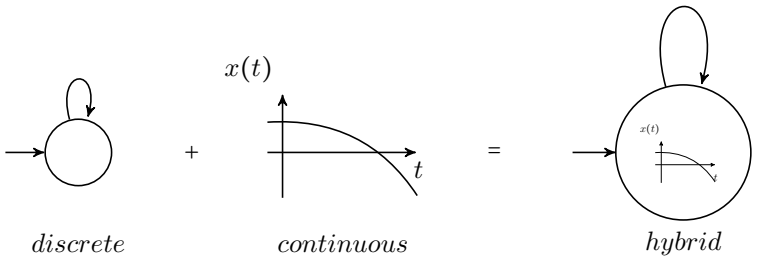


Figure 2.1: Hybrid systems consist of both discrete and continuous components. This cohesion can be intuitively seen as a discrete automaton that shows continuous behaviour in its locations.

2.1 Hybrid Automata

In order to be able to formally specify the reachability problem for hybrid systems, I priorly present a popular modeling formalism for suchlike systems, namely the *hybrid automaton*, according to [ACH+95], and a special subclass, the *linear hybrid automaton*:

Definition 2.1 (Syntax of a hybrid automaton). *A* hybrid automaton $HA = (Loc, Var, Con, Lab, Edge, Act, Inv, Init)$ *is an 8-tuple where*

- Loc *is a finite set of* locations.
- Var *is a finite set of real-valued* variables. *A* valuation *is a function* $v : Var \to \mathbb{R}$ *assigning values to these variables.* V *denotes the set of all valuations for a variable set* Var.
- Con *is a finite set of* controlled variables *with* $Con \subseteq Var$.[1]
- Lab *is a finite set of* synchronisation labels *including the* stutter label $\tau \in Lab$.
- $Edge$ *is a finite set of* transitions *with* $Edge \subseteq Loc \times Lab \times 2^{V^2} \times Loc$.[2] $Edge$ *contains a* τ-transition (stutter transition) *of the form* $(\ell, \tau, Id_{Con}, \ell)$ *for each location* $\ell \in Loc$ *with* $(v,v') \in Id_{Con}$ *iff* $\forall x \in Var$ *either* $x \notin Con$ *or* $v(x) = v'(x)$ *holds.*[3]
- Act *is a function assigning a set of* activities *to each location* $\ell \in Loc$. *An activity is a function* $f : \mathbb{R}_{\geq 0} \to V$. *Every activity is required to be* time-invariant*:* $\forall \ell \in Loc$, $f \in Act(\ell)$ *and* $t \in \mathbb{R}_{\geq 0}$ *has to hold:* $(f + t) \in Act(\ell)$ *where* $(f + t)(t') = f(t + t')$ $\forall t' \in \mathbb{R}_{\geq 0}$. *This means that an activity* f *of a location* ℓ *needs to stay a valid activity of the location without regard of the input time value.*
- Inv *is a function assigning an* invariant $Inv(\ell) \subseteq V$ *to every location* $\ell \in Loc$.
- $Init$ *is a finite set of* initial states *with* $Init \subseteq \Sigma$ *where* $\Sigma = Loc \times V$ *formalises the set of states, i.e. the state of a hybrid automaton is described by a pair* (ℓ,v) *of a location* ℓ *and a valuation* v.

Hybrid automata can both perform discrete steps and continuous time steps each with their own semantics:

Definition 2.2 (Discrete transition-step relation). *The* transition-step relation $\to^a$ *of a hybrid automaton* $HA = (Loc, Var, Con, Lab, Edge, Act, Inv, Init)$ *with current state* (ℓ, v) *and* $a \in Lab$ *for discrete steps is given by the following semantic rule:*

[1] Controlled variables are merely pertinent to the parallel composition of hybrid automata which is of no interest regarding this bachelor thesis. The set Con is therefore solely listed here for the sake of completeness. Interested readers can find a complete specification with [ACH+95]

[2] The employed notation 2^M is a formalisation of the powerset concerning the set M, i.e. 2^M is the set of all subsets P of M: $2^M = \{P \mid P \subseteq M\}$.

[3] Stutter transitions can basically be seen as "do nothing" steps which, similarly to controlled variables, only play a major role in parallel compositions of hybrid automata. They are thus as well without relevance for this thesis which is why I will not further specify them and their purpose. Here, too, you can refer to [ACH+95] for more information.

$$\frac{(\ell, a, \mu, \ell') \in Edge \quad (v, v') \in \mu \quad v' \in Inv(\ell')}{(\ell, v) \rightarrow^{a} (\ell', v')}$$

where $\ell, \ell' \in Loc$ *and* $\mu \subseteq V^2$.

Intuitively a discrete step $(\ell, v) \rightarrow^{a} (\ell', v')$ is feasible if there is an enabled transition (ℓ, a, μ, ℓ') with label a from location ℓ to location ℓ' and the invariant of ℓ' is satisfied after the step. In this context the term "enabled" refers to an existence of a target valuation v' that can be paired with the current valuation v according to the transition relation μ. It is to be noted that the specification for the transformation from v to v' when taking a transition is often called *reset function*.

Definition 2.3 (Time-step relation). *The* time-step relation $\rightarrow^{t}$ *of a hybrid automaton* $HA = (Loc, Var, Con, Lab, Edge, Act, Inv, Init)$ *with current state* (l, v) *for time steps is given by the subsequent semantic rule:*

$$\frac{f \in Act(\ell) \quad f(0) = v \quad f(t) = v' \quad t \geq 0 \quad f([0, t]) \in Inv(\ell)}{(\ell, v) \rightarrow^{t} (\ell, v')}$$

where $\ell \in Loc$, $t \in \mathbb{R}$ *and* $v, v' \in V$.

By intuition the activities describe at which rate the variables change over time when staying in a certain location. A time step $(\ell, v) \rightarrow^{t} (\ell, v')$ is thus only possible if there is an activity f in the current location ℓ that assigns the initial valuation v to the time point 0 and the resulting valuation v' to the time point reached after letting a certain time t elapse. The control is however only allowed to stay in a location ℓ as long as the invariant $Inv(\ell)$ is not violated, i.e. the invariant has to hold for time points 0, t, and all time points in between. Activities are usually given implicitly by *ordinary differential equations (ODEs)*[1] with the corresponding activity being the solution to the equation.

So as to complete the definition of general hybrid automata, at least for my purposes, I additionally define the *derivation relation*, single *execution steps* and the *execution* as a whole for this model:

Definition 2.4 (Derivation relation, execution step and execution).

- *The* derivation relation $\rightarrow$ *of a hybrid automaton* $HA = (Loc, Var, Con, Lab, Edge, Act, Inv, Init)$ *is the union of the transition-step relation and the time-step relation.*
- *A step* $\sigma \rightarrow \sigma'$ *with* $\sigma, \sigma' \in \Sigma$ *is called an* execution step *of* HA.
- *An* execution *(alternatively* path *or* run*)* π *of* HA *is a sequence* $\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 ...$ *with* $\sigma_i \in \Sigma$, $\sigma_i = (\ell_i, v_i)$, $v_0 \in Inv(\ell_0)$, *and* $\sigma_i \rightarrow \sigma_{i+1}$ $\forall i \geq 0$. $\Pi_{HA}(\sigma)$ *denotes the set of all paths of* HA *starting in state* σ. *A state* σ *is reachable iff there is a run of* HA *starting in an initial state* $\sigma_0 \in Init$ *and leading to* σ.

The HyPro project as well as this bachelor thesis both focus on a special form of hybrid automata, the *linear hybrid automata*;[2] the following definitions prepare the formalisation of this specific kind:

[1] An ODE is a differential equation with only one independent variable and its related derivatives.

[2] The term "linear hybrid automaton" is ambiguous in literature: Some definitions require linear hybrid automata to have constant derivatives in terms of activities, others need linear ODEs; the definition presented in this final paper belongs to the latter group.

Definition 2.5 (Linear term, linear constraint, linear set).

- *A* linear term *t over the variable set Var is a linear combination of variables from Var with real-numbered coefficients, i.e. for t holds $t = k_0 + k_1x_1 + k_2x_2 + ... + k_nx_n$ where $x_1,...,x_n \in Var$ and $k_0,...,k_n \in \mathbb{R}$.*
- *A* linear constraint *c over the variable set Var is of the form $t_1 \circ t_2$, where t_1 and t_2 are both linear terms over Var and $\circ \in \{\leq, <, =, >, \geq\}$.*
- *A* linear set *L over the variable set Var is a set defined by Boolean conjunctions of finitely many linear constraints over Var. Note that linear set are not functional complete due to the missing negation. However, negations are restrictedly realisable, as the negation is given partly via the range of the constraint operators $\circ$, e.g. the linear constraint $t_1 \geq t_2$ can be negated by replacing it with the linear constraint $t_1 < t_2$.*

At last I am able to determine linear hybrid automata:

Definition 2.6 (Linear hybrid automaton). *A* linear hybrid automaton *$LHA = (Loc, Var, Con, Lab, Edge, Act, Inv, Init)$ is an 8-tuple where syntax and semantics are identical to those of a general hybrid automata (cf. Definitions 2.1 – 2.4) with the following three additional restrictions:*

- *For every location $\ell \in Loc$, the activities $Act(\ell)$ are described by linear ordinary differential equations.*
- *For every location $\ell \in Loc$, the initial set $Init$, the invariants $Inv(\ell)$ and the guards ψ[1] of the transitions $Edge$ are linear sets.*
- *The variable assignments need to be deterministic for all transitions in $Edge$, i.e. per edge there is only one resulting valuation v' allowed for every initial valuation v concerning pairs $(v,v') \in \mu$.*

Example 2.1 (Smoke detector). *To clarify the concepts defined above, I present a very simple exemplary model of a hybrid system in the following with the depiction of the corresponding linear hybrid automaton available via Figure 2.2:*

A newly bought smoke detector is hanging on the ceiling fully operational, starting out with 100% of its battery charge x in the initial location "normal". While hanging there, the battery charge slowly depletes with a constant factor -1 over time. When the charge falls below 30%, the smoke detector enters a location "warning" in which it constantly emits a beeping noise until the user exchanges the batteries. While remaining in this location, the battery charge drops twice as fast due to the production of the warning sound. In case the user does not replace the batteries, the smoke detector will at last enter the location "depleted" in which it is no longer able to operate.

For reasons of simplicity I assume that the new batteries are always fully charged, i.e. exchanging them brings x back to 100%. It is possible to replace the current batteries at any point.

A formal specification of the related automaton presented in Figure 2.2 is given below:

[1] Guards are used by hybrid automata to determine whether it is allowed to take a transition with the current valuation v, i.e. the transition is only enabled if v satisfies the guard condition. Guards partly define the transition relation μ.

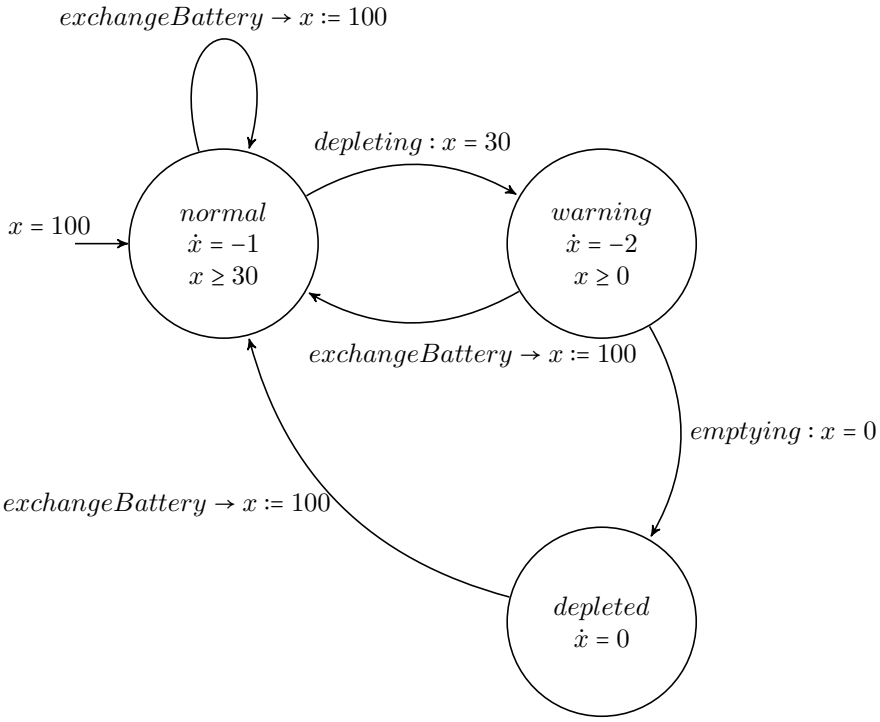


Figure 2.2: The linear hybrid automaton model of the smoke detector.

- $Loc = \{normal, warning, depleted\}$,
- $Var = \{x\}$,
- $Con(normal) = Con(warning) = Con(depleted) = \{x\}$,
- $Lab = \{\tau, exchangeBattery, depleting\}$,
- $Edge =$
  $\{(normal, exchangeBattery, \{(v,v') \in V^2 \mid v'(x) = 100\}, normal),$
  $(normal, depleting, \{(v,v') \in V^2 \mid v(x) = 30 \wedge v'(x) = v(x)\}, warning),$
  $(warning, exchangeBattery, \{(v,v') \in V^2 \mid v'(x) = 100\}, normal),$
  $(warning, emptying, \{(v,v') \in V^2 \mid v(x) = 0 \wedge v'(x) = v(x)\}, depleted),$
  $(depleted, exchangeBattery, \{(v,v') \in V^2 \mid v'(x) = 100\}, normal)\}$,
- $Act(normal) = \{f : \mathbb{R}_{\geq 0} \to V \mid \exists c \in \mathbb{R}. \forall t \in \mathbb{R}_{\geq 0}. f(t)(x) = -1t + c\}$,
  $Act(warning) = \{f : \mathbb{R}_{\geq 0} \to V \mid \exists c \in \mathbb{R}. \forall t \in \mathbb{R}_{\geq 0}. f(t)(x) = -2t + c\}$,
  $Act(depleted) = \{f : \mathbb{R}_{\geq 0} \to V \mid \exists c \in \mathbb{R}. \forall t \in \mathbb{R}_{\geq 0}. f(t)(x) = c\}$,
- $Inv(normal) = \{v \in V \mid v(x) \geq 30\}$,
  $Inv(warning) = \{v \in V \mid v(x) \geq 0\}$,
  $Inv(depleted) = \{true\}$,
- $Init = \{(normal, v) \in \Sigma \mid v(x) = 100\}$.

2.2 Reachability Analysis for Hybrid Systems

Hybrid systems reachability analysis approaches in general aim for solving the *hybrid systems reachability problem*:

Definition 2.7 (Hybrid systems reachability problem). *Given a hybrid automaton HA with an initial state set $Init$ and a target state set P, the* hybrid systems reachability problem *is to decide whether there is a state in P that is reachable from the initial states $Init$ in HA.*[1]

If the set P is a set of unsafe states for the automaton HA, the reachability problem becomes a safety verification problem. The reachability problem for general hybrid automata is undecidable. For some subclasses (e.g. initialized rectangular automata), the problem becomes decidable [HKPV95].

As mentioned before, this work focuses on linear hybrid automata (cf. Definition 2.6) and thus I restrict the reachability problem from now on to only deal with this type of automata. This specialisation simplifies the problem which nonetheless stays undecidable in most cases; however, a good number of incomplete approaches exists for linear hybrid automata and its undecidable subclasses while simple subclasses already have been proven to be decidable (e.g. [AD94, HR98]).

As already stated in Chapter 1, there are three main categories of reachability analysis techniques, one of them utilising *theorem proving* which resulted in proof systems that can be used for substantiating invariants and verification conditions in basically every system class like shown in [AMHS01]. Unfortunately, theorem proving has required interactivity and can usually only handle systems of limited size. A well-known tool that uses theorem proving is KeYmaera [PQ08].

Interval constraint propagation poses another major group of approaches and is in this context usually embedded into *satisfiability modulo theories (SMT) solvers* that make use of first-order logic to verify the safety of the system. A variety of tools regarding interval constraint propagation can be found such as iSAT-ODE [Egg14], dReach [KGCC15], HSolver [RS07] and Ariadne [CBGV12].

Regarding this final paper, the focus lies on *fixed point computations*, the third class of procedures, that generally use geometric and/or symbolic state set representations (such computations were firstly introduced in [ACH+95]).

2.2.1 Fixed Point Computation

Fixed point computations can further be diversified into *forward fixed point computation* and *backward fixed point computation*: Forward fixed point computation gradually computes the set of all reachable states by extending the set of initial states with all possible successing states until no new states are detected anymore, i.e. a fixed point is reached; the procedure then checks for a non-empty intersection with the target state set P to evaluate the safety of the system.

A general forward fixed point computation procedure is formalised with Algorithm 2.1; the algorithm receives an initial state set $Init$ and a target state set P and returns true if the target set P is reachable and false otherwise. It can be easily seen that this procedure does not necessarily terminate if the state space is infinite.

[1] In the future I will refer to the hybrid systems reachability problem simply with "reachability problem".

```
1: procedure BasicForwardReach(Init, P)
2:     R^new ← Init;
3:     R ← ∅;,
4:     while R^new ≠ ∅ do                          ▷ while new states are found
5:         R ← R ∪ R^new;
6:         if R ∩ P ≠ ∅ then
7:             return true;                        ▷ target set is reached
8:         else
9:             R^new ← Reach(R^new);               ▷ extend reachable set
10:            R^new ← R^new ∖ R;                  ▷ identify new states
11:        end if
12:    end while
13:    return false;                               ▷ target set is not reachable
14: end procedure
```

Algorithm 2.1: Basic forward reachability algorithm.

Backward fixed point computation on the other hand starts out from the target set P and gradually computes all possible predecessors of those states until a fixed point is reached; it then reviews intersection with the set of initial states.

Both forms of fixed point computation are suitable choices for reachability analysis, but many computations using any form of fixed point computation do simply not terminate and if they do, they still cost a lot of resources. Even so, backward fixed point computation is not possible in some cases, e.g. in the case of non-invertible reset functions, i.e. indeterministic predecessors. However, indeterministic predecessors are excluded in my definition of linear hybrid automata (cf. Definition 2.6).

The abstract concepts of forward fixed point computation and backwards fixed point computation are contrasted via Figure 2.3.

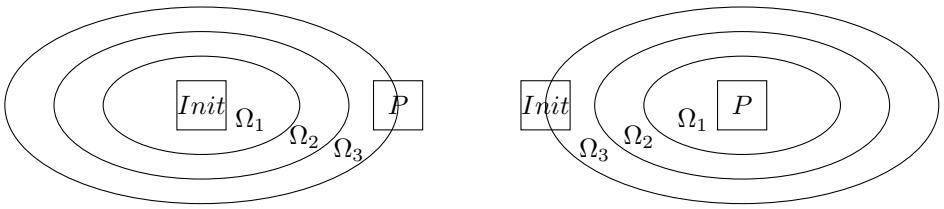


Figure 2.3: Abstract juxtaposition concerning the concepts of forward fixed point computation (left) and backwards fixed point computation (right) with the set of initial states being denoted by $Init$, the set of target states being represented by P. The reachable state set after computation step i is displayed via the set Ω_i.

As representation of the sets is per se difficult, *over-approximations* are often advisable, since they give the possibility to use efficient state set representations and to simplify computation. However, an approximation error is nearly certainly present as a trade-off; when using over-approximations the system can only be proven safe - never unsafe: An intersection with the target/initial set could possibly only be present due to the over-approximation in which case another computation run with a more precise over-approximation could be the next step and may return a different result.

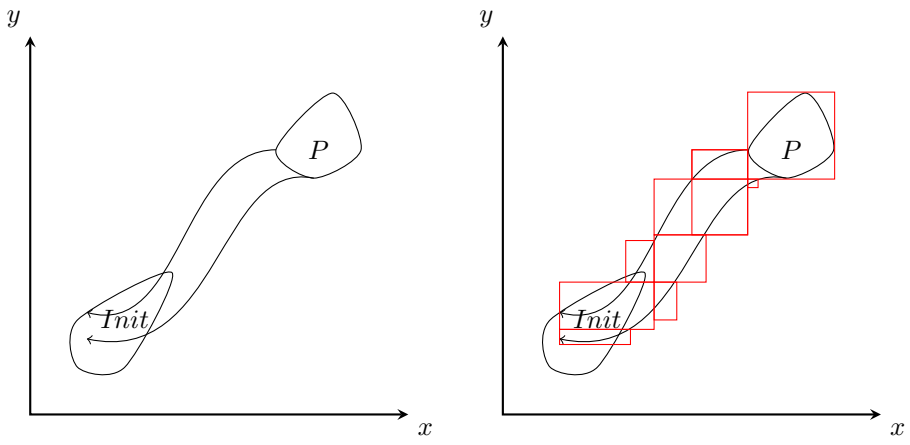


Figure 2.4: Illustration of the initial state set *Init*, the target state set P, the reachable set of predecessors of P which is indicated by arrows (left) and an exemplary over-approximation using hyperrectangles that are displayed in red (right).

A visual example of a backward fixed point computation using hyperrectangles[1] as geometrical over-approximations is given by Figure 2.4. Concerning this example, the computation run detects an intersection with the initial set.

If proving a system as unsafe is the main objective, one could also use *under-approximations* of the reachable state set. The situation here is converse: An intersection with the initial/target set using under-approximations would substantiate the system's unsafety, but could never show the system's safety.

2.2.2 Flow Pipe Construction

It is also possible to use a *flow pipe* construction as a special form of fixed point computation to reduce the approximation error as presented in [CK98]. In fact, HyPro and many fixed point computation tools make use of a flow pipe, such as Flow* [CÁS13], HyCreate [HyC], Cora [AD14] and SpaceEx [FGD+11]. Please note that the several state set operations that concern flow pipe construction and are thus mentioned in this section are formally specified throughout Chapter 3. The content of this subsection is based on [LG09].

The basic idea of the flow pipe construction revolves around considering only a bounded flow duration and dividing this whole so-called *time horizon* into smaller segments (time steps) of length δ, and over-approximating these single time steps separately step-by-step. The resulting over-approximation is given via the union of the over-approximations for each segment.

Regarding the first over-approximatively computed segment Ω_0, the required computational effort is higher than for subsequent segments and the procedure slightly differs based on the *bloating technique* that is used: Without regard of the bloating technique, computation of Ω_0 always begins by computing the reachable state set at time point δ (Ω_δ) via a *linear transformation* of the initial state set *Init*.

[1] Please refer to Chapter 3.2 for a formal definition of hyperrectangles

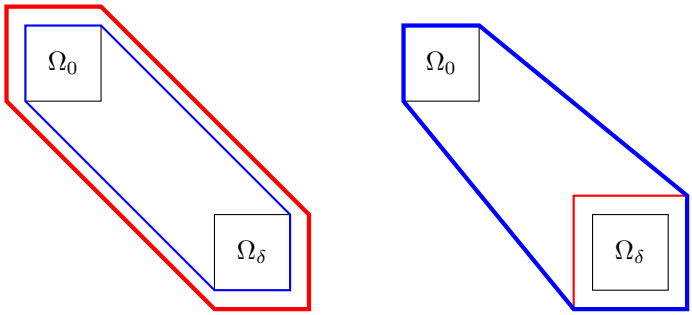


Figure 2.5: Schematic computation of the first flow pipe segment using uniform bloating (left) and improved bloating (right). The initial state set is denoted by Ω_0, the reachable state set at time point δ by Ω_δ. Result sets of the convex hull of the union are displayed in blue, while those of the Minkowski sum are visualised in red.

Afterwards, when utilising *uniform bloating*, the *convex hull of the union* of *Init* and Ω_δ is calculated in an attempt to cover the reachable set in between time points 0 and δ as well. As this is usually not sufficient to cover all possible trajectories of the system, the resulting convex hull is expanded by using a *bloating factor* α[1]: The *Minkowski sum* of the convex hull and usually a ball B of radius α is calculated in case of an autonomous system; for non-autonomous systems, the external influence is described by a factor β and thus the ball B is of a greater radius $(\alpha + \beta)$ to reflect the influence of the external input.

In case of *improved bloating*, the bloating takes place earlier in the computation, right after obtaining the reachable set at time point δ via linear transformation: This newly computed set is bloated before computing the convex hull of the union of the just described set and the initial set. Improved bloating yields smaller over-approximations in the general case which makes it superior to uniform bloating.

HyPro currently supports both uniform and improved bloating. Both concepts are contrasted via Figure 2.5.

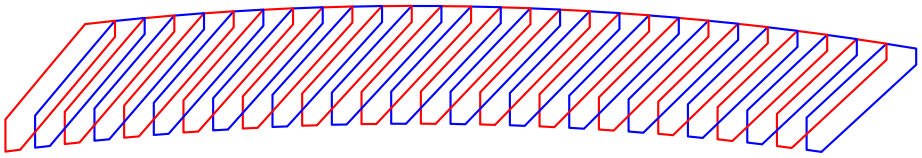


Figure 2.6: Part of an actual flow pipe plot regarding a bouncing ball model. The segments are displayed in alternating colours for reasons of clarity.

Every segment after the first one is simply computed by applying linear transformations on the initial segment Ω_0, resulting in very similar geometry of each segment (cf. Figure 2.6). After computing a segment, the *intersection* of the currently reachable set and existing guard sets and the invariant set of the current location are necessary for a correct analysis of the system. Additionally to that, the *non-emptiness*

[1] The bloating factor α depends on a number of other factors like the initial set and time step size. For more information on this topic, please refer to [Gir05].

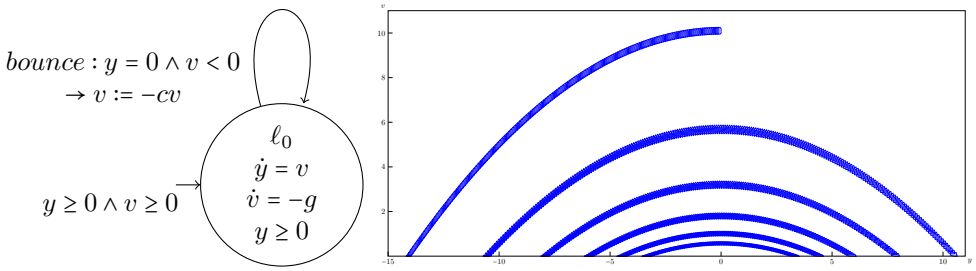


Figure 2.7: Complete flow pipe plot (right) of a bouncing ball model (left) with gravity constant $g = 9.81$ and some other constant $c = 0.75$: The ball starts at height $y \in [10,10.2]$ and starts falling with velocity $v = 0$. In the underlying automaton, a transition to the same state is taken every time the ball hits the ground at $y = 0$ and bounces off. The flow then continues at the right hand side of the graphic and loses velocity and height overall with progressing time. The time horizon is 3, the time step size δ is 0.01 and the maximum number of allowed transition usages is 5.

concerning the intersection of the reachable set and the target set P has to be tested in order to be able to verify properties. Furthermore, in case of non-autonomous systems, additional bloating via Minkowski sum is required after computing a new segment. An exemplary plot of a complete flow pipe computation concerning a bouncing ball model is given by Figure 2.7.

2.3 Controller Synthesis for Hybrid Systems

Reachability analysis is not the only application area for under-approximations of state set representations, it also plays an important role in terms of *controller synthesis* for hybrid systems. Even so, the concept of controller synthesis has no real relevance for this bachelor thesis and is presented briefly for reasons of integrity.

The *switching controller synthesis* is a popular approach [WT97]: Switching controller synthesis aims for designing a controller C of a hybrid automaton HA that ensures the safety of the system during runtime: The controller C supervises the momentary state of HA and forces the execution of controllable transitions of HA in order to prevent the automaton from reaching unwanted or even unsafe states. Under-approximations are in the context of controller synthesis additionally used for creating counterexamples that can be utilised for refining the model [CFH+03].

Chapter 3

State Set Representations

Since the flow pipe construction operates on state sets and, as already implied, uses mainly geometrical objects for over-approximating these sets, I will now take a closer look at the options that are currently available concerning the choice of representation:

There are several options as regards representation when approximating the reachable sets of hybrid systems all of which can be categorized by two main classes of state set representations, namely *geometric* and *symbolic state set representations*:

Geometric state set representations, as one would expect, revolve around utilising multidimensional geometric objects in order to represent the reachable sets, the most popular ones being *convex polyhedra*, *orthogonal polyhedra*, *hyperrectangles*, *zonotopes* and *ellipsoids*.

Alternatively symbolic state set representations are much more abstract, often relying on algebraic approaches. The current main representatives here are *support functions* and *Taylor models*.

With so many different representations being used and experimented with, it has yet proven difficult to find something like an optimal state set representation, since each representation comes with its very own set of advantages and disadvantages (cf. Section 3.6). Choosing a suitable representation or even multiple representations for one's own purposes depends on many factors and usually a compromise is to be made:

In general, the more complex the state set representation chosen, the more memory space is needed, the more intricate it is to perform necessary operations for reachability analysis (like intersection and union), and thus the more costly it is to approximate the reachable set; however, the precision of the approximation highly improves at the same time.

The HyPro project currently supports convex polyhedra, hyperrectangles, zonotopes and support functions which is why I focus on these four representations. In the following sections, I define them and give fitting examples. Although not being of main interest for this bachelor thesis, I still present orthogonal polyhedra, ellipsoids and taylor models afterwards shortly in Section 3.5, before ending this chapter with Section 3.6 which takes a closer look at the necessary operations on state sets during reachability analysis and especially on how well they can be performed on these distinct representations. Most of the content in this chapter is based on [LG09].

3.1 Convex Polyhedra

Convex polyhedra (e.g. [Zie95]) are, as one would expect, a special form of *polyhedra*, closer specified via the following two definitions:

Definition 3.1 (Polyhedron, polytope). *A* polyhedron PH *is a solid object in the d-dimensional Euclidian space and the solution set to a finite number of inequalities with real coefficients and d real variables. A* polytope *is a bounded polyhedron.*[1]

Definition 3.2 (Convex polyhedron, convex set, convex polytope). *A* convex polyhedron *is a polyhedron PH which set of points is also a* convex set *in the d-dimensional Euclidian space, i.e. for every possible pair of points p_1, p_2 from PH must hold that every point that lies on the straight line segment between p_1 and p_2 is also a point of PH:*

$$\forall p_1, p_2 \in PH. \forall \lambda \in [0,1] \subseteq \mathbb{R}. \lambda p_1 + (1-\lambda) p_2 \in PH.$$

A convex polytope *is a bounded convex polyhedron.*

Convex polyhedra in contrast to non-convex polyhedra provide benefits regarding the representation possibilities of the objects and thus simplify many operations that are usually conducted on these solids.[2] Nevertheless, non-convex polyhedra are still rarely being utilised for reachability analysis, mainly in the form of orthogonal polyhedra (cf. Section 3.5).

There are commonly two different representations being used for convex polyhedra, namely the *H-representation* and the *V-representation*. In order to be able to define the H-representation, I define *hyperplanes* and *half-spaces* beforehand:

Definition 3.3 (Hyperplane). *A d-dimensional* hyperplane *is a subspace that is of one dimension less than the surrounding ambient space.*

Hyperplanes are often described by a (non-zero) *normal vector* c of dimension d that is orthogonal to the hyperplane and defines the alignment, and an *offset* z which specifies the distance from the origin to that plane.[3] A hyperplane can also be written as a linear equality.

Definition 3.4 (Half-space). *A d-dimensional* half-space H *is that part of a d-dimensional Euclidian space obtained by removing the part lying on one side of a $(d-1)$-dimensional hyperplane. A half-space H is* closed *iff H contains its space-dividing hyperplane. Otherwise H is called an* open *half-space.*

A half-space can also be written as a linear inequality and can as well as hyperplanes be defined by a (non-zero) normal vector c and an offset z of the corresponding space-dividing hyperplane. For future reference I will presume that the normal of this space-dividing hyperplane is always pointing outwards in relation to the related half-space, unambiguously defining it.

[1] Please note that the terms "polyhedra" and "polytope" are also ambiguous in literature: A polyhedron is often defined as a three-dimensional object *only*; in case of that definition, a polytope could be formalised as the generalisation of a polyhedron for dimensions $d > 3$ and could thus also be undbounded.

[2] For more information on convex sets, convex polyhedra and their special properties, you can refer to [Gal08].

[3] As the term of the "offset" sometimes refers to a distance value in a non-normalised plane equation, I personally refer to the Euclidian distance with this term, i.e. I expect the normal vector c to be normalised such that the offset always represents the Euclidian distance from hyperplane to origin.

Definition 3.5 (*H*-polyhedron, *H*-polytope). *A d-dimensional* H-polyhedron HP *is represented by the intersection of finitely many closed d-dimensional half-spaces* H_i *(yielding the* facets *of the polyhedron), i.e.* $HP = \bigcap_{i=1}^{n} H_i$, *where* $n \in \mathbb{N}_{>0}$.

A H-polytope *is a bounded H-polyhedron.*

It is common to specify an H-polyhedron by a matrix inequality of the form $HP = \{x \in \mathbb{R}^d \mid Cx \leq z\}$, where C is a matrix containing the normals of the half-spaces as rows and z is a vector comprising the constant distances of the associated half-spaces from the origin. This yields the H-representation:

Definition 3.6 (*H*-representation). *An* H-representation *of a polyhedron that is defined by* m *halfspaces is a tuple* (C,z) *with* normal matrix $C \in \mathbb{R}^{m \times n}$ *containing rows* c_i *which represent the associated half-space normals, and* distance (offset) vector $z \in \mathbb{R}^m$.

There is an infinite amount of H-representations for every single polyhedron, as the representation may comprise redundant half-spaces; however, for every full-dimensional convex polyhedron exists a minimal unique H-representation.

While H-polyhedra are defined via their facets, *V-polyhedra* are formalised by the convex hull of their vertices:

Definition 3.7 (Convex hull). *The* convex hull $CH(V)$ *of a set* $V \subseteq \mathbb{R}^d$ *is the smallest convex set that contains* V. *For a finite set* $V = \{v_1, ..., v_n\}$, *the convex hull is given via*

$$CH(V) = \{x \in \mathbb{R}^d \mid \exists \lambda_1, ..., \lambda_n \in [0,1] \subseteq \mathbb{R}^d . \sum_{i=1}^{n} \lambda_i v_i = x \wedge \sum_{i=1}^{n} \lambda_i = 1\}.$$

Definition 3.8 (*V*-polyhedron, *V*-representation, *V*-polytope).
A d-dimensional V-polyhedron VP *is the convex hull of a finite set* $V \subset \mathbb{R}^d$, *i.e.* $VP = CH(V)$. *The elements of the set* V *are called the* vertices *of* VP. *The set* V *is called a* V-representation *of* VP.

A V-polytope *is a bounded V-polyhedron.*

Similarly to the H-representation, V-representations may contain redundant vertices, as it is possible to add inner points of the polyhedron as vertices. This and the possibility to intersect infinitely many half-spaces in case of an H-representation leads to the fact that there is no constant representation size for polyhedra, which may demands for reduction algorithms during computation in order to remove redundant (in terms of implementation) half-spaces or vertices.[1] It is worth noting that it is not possible to construct an unbounded V-polyhedron when using the above presented definition (other definitions can allow unboundedness[2]).

Both representations are commonly used, since some operations are easier with an H-representation of the object while others are easier with a V-representation (cf. Section 3.6). An exact conversion into the favourable representation for each operation is often not advised, as the exact computation of the complementary representation

[1] An example for such a reduction algorithm is presented in Subsection 4.2.2 of this thesis.

[2] Definitions that allow for unboundedness usually permit the usage of polyhedral cones for the construction of the object.

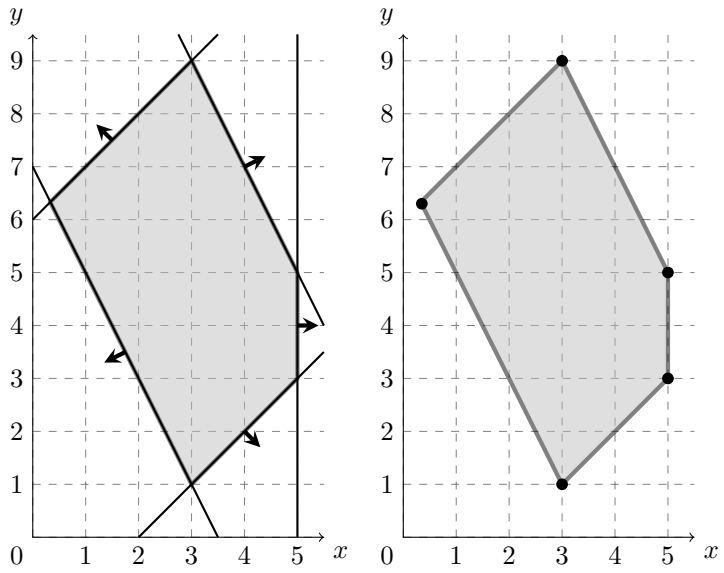


Figure 3.1: A 2-dimensional convex polytope in minimal representation (visualised by a gray area), once displayed as the intersection of five half-spaces with corresponding normals (left) and once represented by the convex hull of its five vertices (right).

usually has exponential cost. In a general sense, convex polyhedra provide precise approximations but are costly to handle.

A visual example of a 2-dimensional convex polytope with the underlying minimal information stored for each representation can be seen in Figure 3.1.

3.2 Hyperrectangles

Hyperrectangles (e.g. [MKC09]) form a very simple type of state set representation:

Definition 3.9 (Hyperrectangle). *A d-dimensional* hyperrectangle *(or* box*) B is the cross product of d real-numbered intervals.*

This simple definition yields a few interesting properties:

- Every box has $2d$ facets and 2^d vertices due to the fixed number of d defining intervals. Complexity of the representation thus only scales linearly with the dimension d.

- Most operations can be done very efficiently on hyperrectangles because of their simple structure. However, they are not closed regarding the majority of operations (cf. Section 3.6).

- Each edge of any box is parallel to one coordinate axis which results in a fixed alignment in space. This property and the limited number of facets and vertices makes approximations with hyperrectangles in general very unprecise.

In order to minimise the approximation error when approximating a convex set Ω with boxes, it is recommended to compute the *interval hull* $\square(\Omega)$ of Ω:

Definition 3.10 (Interval hull). *The* interval hull $\square(\Omega)$ *of a set* Ω *is its smallest enclosing box, i.e.*

$$\square(\Omega) = [\underline{x_1}, \overline{x_1}] \times ... \times [\underline{x_d}, \overline{x_d}]$$

where $\forall i$ *with* $1 \leq i \leq d, \underline{x_i} = \inf\{x_i \mid x \in \Omega\}$ *and* $\overline{x_i} = \sup\{x_i \mid x \in \Omega\}$ *holds.*[1].

An example of the interval hull is illustrated in Figure 3.2.

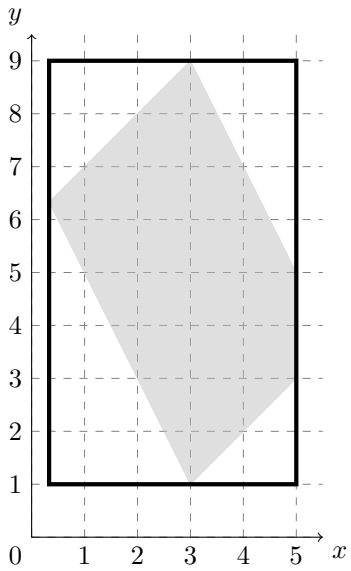


Figure 3.2: The interval hull $\square(\Omega)$ (displayed by thick lines) of the convex set Ω (depicted via the gray area), formally specified with $\square(\Omega) = [0.33,5] \times [1,9]$.

3.3 Zonotopes

The class of *zonotopes* (e.g. [GNZ03])is a special sub-class of convex polyhedra that utilises the *Minkowski sum*:

Definition 3.11 (Minkowski sum). *The* Minkowski sum $X \oplus Y$ *of two sets* X *and* Y *is the set of sums of all possible element pairs from* X *and* Y *by taking one element from each set for every pairing:*

$$X \oplus Y = \{x + y \mid x \in X \wedge y \in Y\}.$$

Figure 3.3 shows an paradigmatic Minkowski addition.

Zonotopes can intuitively be seen as the Minkowski sum of a finite amount of line segments, a more formal definition featuring a *centre point* c and a set of *generators* G which represents these line segments is given hereafter:

[1] x_i is the ith component of x

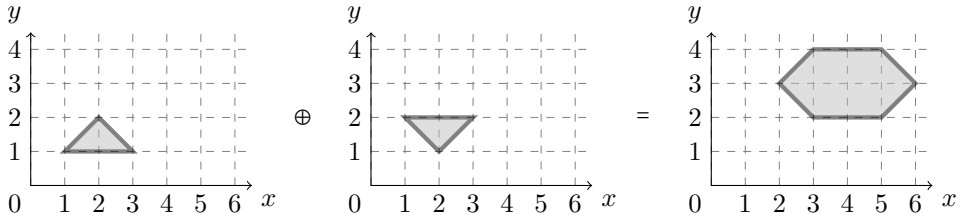


Figure 3.3: The Minkowski sum of two triangles.

Definition 3.12 (Zonotope). *A d-dimensional* Zonotope Z *with a centre point* $c \in \mathbb{R}^d$ *and a set of generators* $G = \{g_1, ..., g_n\}$*, where* $g_i \in \mathbb{R}^d$*,* $n \in \mathbb{N}_{>0}$ *and* $1 \leq i \leq n$*, is given via the following equation:*

$$Z = \{x \in \mathbb{R}^d \mid \exists \alpha_1, ..., \alpha_n \in [-1, 1].x = c + \sum_{i=1}^{n} \alpha_i g_i\}.$$

It is to be noted that as the coefficients $\alpha_1, ..., \alpha_n$ range from -1 to 1, every zonotope contains not only its generators $g_1, ..., g_n$ but also implicitly their corresponding inverse line segments, resulting in a point symmetric set that has its centre point as the centre of symmetry. Furthermore, not only the direction of the generators matter, but also their length. A common way of representing a zonotope Z is by using the notation $< c, g_1, ..., g_n >$. This notation is called the $G - representation$ of Z.

Similarly to polytopes, zonotopes are variable in representation size: It is always possible to add additional generators to G. Zonotopes provide generally a good accuracy in most applications, they are nevertheless limited in this regard due to their centre of symmetry and because of that strictly symmetrical structure. They can be seen as a compromise between polyhedra and hyperrectangles, as they supply potentially better approximations than boxes, while lacking precision in comparison to polyhedra that can also express non-symmetrical objects. So as to support the rather abstract definition of zonotopes, a step-by-step construction of an exemplary zonotope can be seen in Figure 3.4.

3.4 Support Functions

While all previously presented representations are determined by a set of parameters geometrically, *support functions* (e.g. [GG09]) symbolically represent the underlying object as a mathematical function:

Definition 3.13 (Support function). *The* support function p_Ω *of a convex set* $\Omega \subseteq \mathbb{R}^d$ *is a function defined as follows:*

$$p_\Omega : \mathbb{R}^d \to \mathbb{R} \cup \{\infty, -\infty\}$$

$$\ell \mapsto \sup_{x \in \Omega} x \cdot \ell$$

Intuitively, evaluating the support function p_Ω for an input vector ℓ and a convex object Ω gives you an indication about where to fittingly place a hyperplane with maximum distance to the origin and normal vector ℓ such that it touches Ω: The

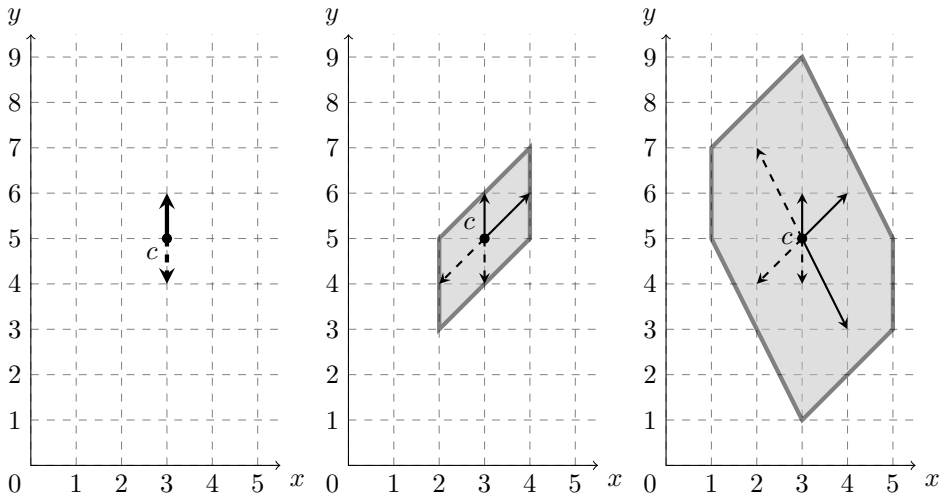


Figure 3.4: A step-by-step construction of a paradigmatic zonotope with center $c = (3,5)^T$ and generators $g_1 = (0,1)^T$, $g_2 = (1,1)^T$ and $g_3 = (1,-2)^T$. Step 1 (left) holds only g_1 and its inverse, step 2 (middle) adds g_2 to the object and step 3 (right) displays the full zonotope by adding the final generator g_3. The inverse of the generators are displayed as dashed vectors while the resulting set per step is visualised via thick lines and a gray area.

result value $p_\Omega(\ell)$ is thus the distance from the origin to mentioned hyperplane (and thus also to the corresponding facet of Ω), since the support function returns the product of the direction ℓ and the outermost point of Ω regarding this direction; the resulting touching hyperplane that is orthogonal to ℓ with offset $p_\Omega(\ell)$ is called a *supporting hyperplane* of Ω. If the underlying object has no outer bound in direction ℓ, the support function returns ∞ or $-\infty$ depending on the chosen direction ℓ. For a better understanding, an exemplary evaluation of a support function in a paradigmatic direction ℓ and its resulting supporting hyperplane are visualised in Figure 3.5.

It is worth noting that closed convex sets Ω are uniquely determined by their support functions as shown by the following equation:

$$CH(\Omega) = \bigcap_{\ell \in \mathbb{R}^d} \{x \in \mathbb{R}^d \mid x \cdot \ell \leq p_\Omega(\ell)\}$$

This means that any convex set Ω, represented by a support function, is the intersection of an infinite set of half-spaces with normal vector ℓ and offset $p_\Omega(\ell)$. A support function can thus be seen as an H-polyhedron defined via an innumerable number of constraining half-spaces.

Computing the value $p_\Omega(\ell)$ for an input vector ℓ means solving a linear optimisation problem: The objective is to maximise the linear function $x \cdot \ell$ for all points $x \in \Omega$. This can be done by using e.g. a linear optimiser.

While most operations that are relevant for reachability analysis can be performed efficiently on support functions and while they offer computational possibilities that geometrical representations cannot serve with (cf. Subsection 3.6.1), they are not without their own shortcomings: When representing convex objects by their support

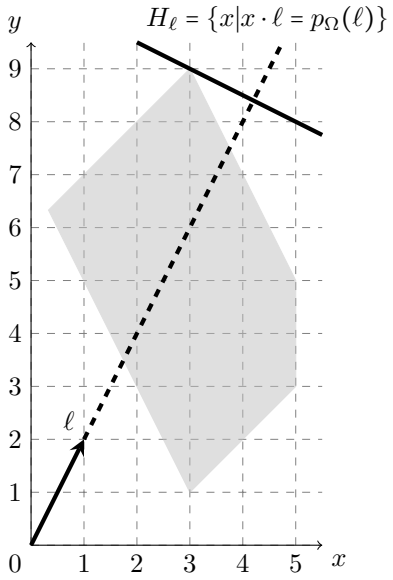


Figure 3.5: A convex set Ω (depicted by the gray area) and its supporting hyperplane with normal vector $\ell = (1,2)^T$.

function, it is hardly possible to derive any information about the geometry without evaluating the function which is usually costly if conducted in vast numbers. Choosing sensible directions ℓ for evaluation purposes can thus have a large impact on the precision of the approximation and computation time by reducing the necessary number of evaluations.

3.5 Other State Set Representations

In this section I give a short introduction to some alternative state set representations that are without further relevance for this bachelor thesis, since they are not supported by HyPro. Nonetheless, these representations are being used in various reachability analysis techniques which is enough to justify their entry here:

- An *ellipsoid* (e.g. [KV00]) is a geometric state set representation that is intuitively the result of transforming an Euclidian ball with an invertible linear transformation. They are usually represented by a *center point* c and a positive definite *shape matrix* $Q = AA^T$. Ellipsoids have a constant representation size (per dimension).

- Another way of representing state sets geometrically is using *orthogonal polyhedra* (e.g. [BMP99]): These type of polyhedra are generally non-convex and are given as the union of a number of special hyperrectangles that are called *elementary boxes*. Contrary to convex polyhedra, there are three possible representations for orthogonal polyhedra, namely the *vertex representation*, the *neighborhood representation* and the *extreme vertex representation*. Similarly to convex polyhedra, the representation size is not limited.

- *Taylor models* (e.g. [CÁS13, Neu]) depict an additional symbolic state set representation: Taylor models are based on taylor expansions and interval arithmetics. They are specified via a d-dimensional *polynomial* p and two *interval domains* $D, I \subseteq \mathbb{R}^d$. The taylor models of order 0 are similar to interval products, while taylor models of order 1 are similar to zonotopes.

3.6 Operations on State Sets

To further explain the usage of so many distinct state set representations regarding hybrid systems reachability analysis, I will take a closer look at how well the different representations perform concerning the main operations that are applied on state sets during flow pipe construction:

These already previously mentioned main operations (cf. Subsection 2.2.2) are the convex hull of the union of two sets, the Minkowski sum of two sets, the intersection of two sets, the non-emptiness test of a set and the linear transformation of a set. Assume that B and C are two subsets of an arbitrary domain and d is the dimension:

- The convex hull (cf. Definition 3.7) of the union $CH(\cdot \cup \cdot)$ is given via

  $$CH(B \cup C) = conv\{x \mid x \in B \vee x \in C\}.$$

  It is used instead of a simple union operation, as convex sets are not closed under union alone. The convex hull of the union is mainly used for computing the first segment of a flow pipe.

- The Minkowski sum $\cdot \oplus \cdot$ (cf. Definition 3.11) is required for bloating.

- The intersection $\cdot \cap \cdot$ is defined as

  $$B \cap C = \{x \mid x \in B \wedge x \in C\}.$$

  Applications include the calculation of the intersection of the current reachable state set with guard, invariant and target sets. Intersection is thus used very frequently.

- The non-emptiness test $\cdot \neq \varnothing$ for a set B simply revolves around checking whether $B = \varnothing$ holds and is required for proving intersections with the target set to be non-empty/empty.

- The linear transformation $\cdot \to_A$ with transformation matrix A (for e.g. rotation and/or scaling) is utilised for obtaining each flow pipe segment during computation. Although not being a linear transformation in the classical sense, the affine transformation that is the translation of an object is also used for flow pipe construction. The *translation vector* b is in this case included via one additional matrix dimension.

For polyhedra, support functions, hyperrectangles and zonotopes, the computational aspects regarding these five operations are summarised in Table 3.1. It can be seen that the computational difficulty of the mentioned operations highly differs with changing representations. Furthermore it is also apparent that no representation provides an efficient procedure for every single one of those operations.

Convex object	Representation	$CH(\cdot \cup \cdot)$	$\cdot \oplus \cdot$	$\cdot \cap \cdot$	$\cdot \neq \varnothing$	$\cdot \rightarrow_A$
Convex polyhedra	H-representation	hard	hard	easy	easy	easy
	V-representation	easy	easy	hard	easy	easy
Zonotopes	G-representation	–	easy	–	easy	easy
Hyperrectangles	constraining intervals	–	easy	easy	easy	–
Support functions	$p : \mathbb{R}^d \rightarrow \mathbb{R} \cup \{\infty, -\infty\}$	easy	easy	hard	hard	easy

Table 3.1: Computational aspects of the representations. An entry "easy" is to be interpreted that there exists a polynomial-time algorithm (concerning dimension and representation size) for the corresponding representation and operation, "hard" stands for the lack thereof. The symbol – denotes that the associated representation is not closed under the proper operation.

Implementations of the above-mentioned operations can be adjusted in their own special fashion for every representation, utilising the special properties of each representation type. However, this is mostly of no concern for this bachelor thesis that focuses on conversion procedures; for the understanding of most of the conversion algorithms presented in Chapter 4 it suffices to know about the mere existance of these implementations. An exception to this is the HyPro implementation of the support function which happens to be a necessary foundation for Section 4.6. Because of that, I make a supplement to operations on support functions and corresponding realisations in the following before concluding this here chapter.

3.6.1 Operations on Support Functions

Current developments in the field of reachability analysis for hybrid systems show an increasing interest concerning support functions in comparison to traditional geometric representations. The reason for this is for the most part that in general only certain areas of a resulting state set are of interest for safety verification. E.g. when the objective revolves around detecting possible intersections of the reachable set with a target set P, evaluating the reachable set in some well chosen directions in case of a representation via support function could make a complete computation of the reachable set superfluous which again saves resources.

The related implementation of HyPro regarding operations on support functions is mostly based on the equations below (cf. [GG09]):

Proposition 3.1. *For all compact convex sets $\Omega, \Omega' \subseteq \mathbb{R}^d$, all matrices A, all positive scalars λ and all vectors $\ell \in \mathbb{R}^d$:*

$$\begin{aligned} p_{A\Omega}(\ell) &= p_\Omega(A^T \ell) \\ p_{\lambda\Omega}(\ell) &= \lambda p_\Omega(\ell) = p_\Omega(\lambda \ell) \\ p_{\Omega \oplus \Omega'}(\ell) &= p_\Omega(\ell) + p_{\Omega'}(\ell) \\ p_{CH(\Omega \cup \Omega')}(\ell) &= \max(p_\Omega(\ell), p_{\Omega'}(\ell)) \end{aligned}$$

HyPro stores one underlying convex object(more specifically an H-polyhedron or ball) per support function Ω at the start of the computation and then memorises the operations conducted on this starting object Ω without actually transforming the object itself. By doing so, when the support function is then evaluated in some direction ℓ, it is sufficient to apply the memorised operations recursively according to Proposition 3.1 in *reverse order* on the normal vector ℓ *only* to determine the

primary direction ℓ' that needs to be evaluated in the starting object Ω in order to get a correct result for the resulting geometry.

In case of the application of binary operations like the Minkowski sum it is required to "backtrack" these operations with directions ℓ recursively to *all* underlying objects that are part of the sequence of performed operations and not just one object, e.g. to determine the correct direction ℓ' for a convex set $\Omega = \Omega_1 \oplus \Omega_2$ represented by a support function, it is required, when using this method, to evaluate once for Ω_1 and once for Ω_2 and to sum both result values up afterwards.

As a matter of fact, the more detailed handlings of the main operations on support functions by HyPro in each step, excluding the intersection and the non-emptiness test which are not given in the form of equations by Proposition 3.1, are listed in the following:

- The linear transformation $\cdot \rightarrow$ of a set is simply handled by multiplying the transformation matrix A in its transposed form with input direction ℓ. In the case of A being a rotation matrix, the rotation gets inverted, as for rotation matrices $A^T = A^{-1}$ holds. It is to be noted that affine transformations on support functions are computable as well, although they need more computational effort and are not very important regarding reachability analysis.

- When trying to evaluate an object in an input direction ℓ that is the result of a Minkowski sum $\cdot \oplus \cdot$, the computation branches into the operation history of both operands and sums the evaluation results from both recursions up.

- In case of encountering the convex hull of a union $CH(\cdot \cup \cdot)$ in a recursion step, the computation descends into both operation histories as well, but computes the maximum of both results in this scenario.

An example that clarifies this procedure in the simple case of evaluating a support function that memorised a single rotation is given in the following:

Example 3.1 (Rotation). *A convex set Ω, represented by a support function, has a 90 degrees rotation around the origin as only operation memorised and shall be evaluated in direction $\ell = (3, 0)^T$ (cf. Figure 3.6).*

In order to evaluate the support function properly, the correct direction for an evaluation in the underlying initial object has to be determined which is achieved by applying the inverse rotation on the direction vector ℓ that yields $\ell' = (0, -3)^T$.

Afterwards, $p_\Omega(\ell')$ and thus implicitly also the hyperplane $H_{\ell'}$ is computed and the correct offset (which amounts to 3 after normalisation) for the hyperplane H_ℓ with normal ℓ is found.

The situation for the intersection $\cdot \cap \cdot$ and the test for emptiness $\cdot \neq \varnothing$ proves to be different: Both operations are very hard to perform on support functions because of the missing information about the underlying geometry, which is why the intersection is merely approximated by HyPro in the general case and the test for non-emptiness is currently not supported at all.

It is thus necessary to over-approximate the support function with a different representation in order to be able to conduct a reliable non-emptiness test. An over-approximation of the intersection is computed by recursively descending into the operation history of the operands and taking the minimum of both evaluations. However, an empty intersection cannot be detected this way in the general case which is why converting into another representation is advised as well in this scenario.

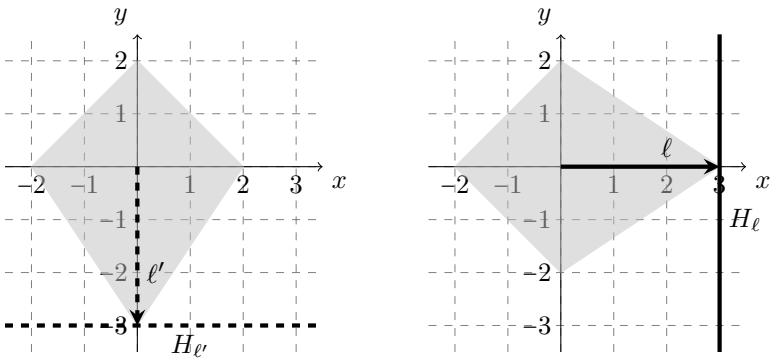


Figure 3.6: The convex set Ω (visualised by a gray area), before the 90 degrees rotation (left) and after the rotation (right) with vectors ℓ and ℓ' and the corresponding supporting hyperplanes H_ℓ and H'_ℓ. The rotation of the whole object is not computed in reality and only displayed for explanatory purposes.

It can also be of advantage to change representation from support function to a different state set representation from time to time, even when not performing these two operations. The reason for this is simply that the list of memorised operations simply grows too large after conducting a lot of operations on the object, which often results in a lot of recursive function calls; converting the support function in an over-approximative manner into a distinct representation and later back to a support function "resets" this memorisation and may improve computation time of future evaluations. Interested readers may wonder how this can be done efficiently; this question leads us to the next chapter of this work which deals with several conversion approaches, also including conversions from various representations to support functions and vice versa.

Chapter 4

Conversion Procedures

Now that the background is covered and the relevant state set representations are defined, I am at last able to present the results of my bachelor thesis, the conversion algorithms, in depth:

In the context of this final paper, I developed nineteen fully operational transformation procedures and two conversion approaches that work limitedly; the underlying issues are discussed in Subsection 4.6.1 of this chapter.

As a part of HyPro, a project that is nearing its completion, most of the code infrastructure and the vast majority of state set operations, basic utility functions, arithmetic operations and datastructures were already programmed without my doing; aside from some code refactoring, the mentioned nineteen conversion approaches and a host of major utility functions for that matter (including a variant of principal component analysis) were implemented by myself and the main focus of my labour. Exact conversion from Zonotope to V-Polyhedron and exact conversion from H-Polyhedron to V-Polyhedron and vice versa were already realised before I commenced my work and were therefore not my doing; the corresponding algorithms are nevertheless covered in this thesis in an attempt to achieve full coverage of the topic.[1]

All dimensions of the Euclidian space are supported by the implementations, however, computing in higher dimensions comes with an expected heavy computation time penalty. The current dimension is always denoted by d throughout this chapter and the index i is an integer with $1 \leq i \leq d$.

From now on, I will make use of the following notation for denoting the *conversion modes*:

- An exact conversion is formalised by a tailed arrow $\rightarrowtail$, and is only successful if both the input object and the output geometry describe exactly the same convex set.

- When referring to an over-approximative conversion, i.e. a transformation that is allowed to produce objects which depict exactly the input sets or even larger sets, the notation is given by a double-headed arrow $\twoheadrightarrow$. A "larger" produced object means in this context that the original object is a proper subset of the resulting object.

[1] All figures in this chapter are artificially constructed example graphics for explanatory purposes and are no actual output plots of my functions. For literal plots, please refer to Chapter 5.

- Under-approximative conversions aim for smaller sets, i.e. sets that are subsets of the original objects, and are represented by dashed arrows ⇢.

The altogether twenty-four conversion approaches are presented in the form of five sections, one for each covered state set representation. Every section deals with those algorithms that transform the corresponding state set representation into a different representation. I tackle the five utilised representations in the same order as they were presented in Chapter 3, with the exception of the V-polyhedra being dealt with even before H-polyhedra, since conversions of V-polyhedra are of high relevance for nearly all other groups of presented algorithms. However, before delving into the conversion algorithms themselves, I introduce a special application of *principal component analysis*, namely the *oriented rectangular hulls (ORHs)*, as they are utilised by some of the developed algorithms.

4.1 Principal Component Analysis and ORHs

Principal component analysis, in short PCA (e.g. [Dun89]), is a mathematical procedure for structuring and simplifying given sets of data by finding the dominating correlations between these sets, allowing for a replacement of the original data with a much smaller, but still representative set.

Olaf Stursberg and Bruce H. Krogh presented a paper (cf. [SK03]) that describes the concept of computing oriented rectangular hulls using PCA. An ORH can intuitively be seen as a hyperrectangle that is aligned in such a way that it encloses the underlying set of data (e.g. a set of vertices) in an optimal fashion. ORHs are therefore due to their orientation in general no longer hyperrectangles by definition, but still retain the beneficial property of featuring the same limited representation size as their strictly axis-oriented counterparts.

Formally, ORHs are defined as the resulting object of the procedure described below; the term of the oriented rectangular hull is therefore bound to that computation method.

Being entirely based on the above-mentioned work by Stursberg and Krogh, I now explain my implementation of principal component analysis that computes the oriented rectangular hull of a forwarded set of *sample points* $X = \{x^1, ..., x^p\}$ in six steps:

1. The first conducted measure is computing the arithmetic mean x^m of X in order to have a point with central tendencies[1] concerning X:

$$x^m = \frac{1}{p} \sum_{j=1}^{p} x^j$$

2. Secondly, the cloud of sample point is (approximately) centered around the origin with the arithmetic mean point positioned exactly at the origin. This is achieved by calculating a set of *translated samples* $\overline{X} = \{\overline{x}^1, ..., \overline{x}^p\}$ with $\overline{x}^j = x^j - x^m$.

[1] The arithmetic mean of a set of points describes in general not the centre point of the set, but merely an approximation of the centre point.

3. The next step is the construction of the *sample matrix* S which simply holds the translated sample points as columns and is formalised as

$$S = \begin{pmatrix} \overline{x}_{1,1} & \cdots & \overline{x}_{1,p} \\ \vdots & \ddots & \vdots \\ \overline{x}_{d,1} & \cdots & \overline{x}_{d,p} \end{pmatrix},$$

where $\overline{x}_{i,j} = x_i^j - x_i^m$ is the ith component of the jth sample point.

4. Subsequently, the *sample covariance matrix* $Cov(S)$ is computed: For two components $\overline{x}_i = x_i - x_i^m$ and $\overline{x}_k = x_k - x_k^m$, the *sample covariance* $Cov(\overline{x}_i, \overline{x}_k)$, which serves as an estimation of the *covariance* between the two vector components, is defined as

$$Cov(\overline{x}_i, \overline{x}_k) = \frac{1}{p-1} \sum_{j=1}^{p} \overline{x}_{i,j} \cdot \overline{x}_{k,j}.$$

The covariance, as a measure of the correlation of two variables, describes in this scenario how the positions of the sample points concerning the two considered dimensional components relate to each other. Therefore, the symmetric sample covariance matrix is filled with the sample covariances regarding all possible pairings of dimensional components:

$$Cov(S) = \begin{pmatrix} Cov(\overline{x}_1,\overline{x}_1) & \cdots & Cov(\overline{x}_1,\overline{x}_d) \\ \vdots & \ddots & \vdots \\ Cov(\overline{x}_d,\overline{x}_1) & \cdots & Cov(\overline{x}_d,\overline{x}_d) \end{pmatrix}$$

For the set of sample points X, $Cov(S)$ represents the distribution of these points in the d-dimensional Euclidian space. The sample covariance matrix is, in terms of my implementation, computed with the following simplified equation:

$$Cov(S) = \frac{1}{p-1} \cdot S \cdot S^T$$

5. Before being able to derive a suitable orientation of the oriented rectangular hull, a *singular value decomposition*[1] needs to be performed on the sample covariance matrix:

$$Cov(S) = U \cdot \Sigma \cdot V^T$$

Both $U \in \mathbb{R}^{d \times d}$ and $V \in \mathbb{R}^{d \times d}$ are unitary matrices and because of the symmetry of $Cov(S)$, $U = V$ holds. The matrix Σ is a diagonal matrix containing the *singular values* σ. Regarding the construction of the ORH, only the matrix U is utilised (or alternatively V). It is worth noting that the singular value decomposition is not necessarily unique.

6. With the matrix U at hand, an enclosing ORH can be derived: Every column of U, where $U_{\bullet,i}$ denotes the ith column of U, defines two half-spaces H_i^+ and H_i^- of the ORH, yielding a total of $2d$ facets:

$$H_i^+ = \{x \mid U_{\bullet,i}^T \cdot x \leq \max_{\overline{x} \in \overline{X}} \{U_{\bullet,i}^T \cdot \overline{x}\} + U_{\bullet,i}^T \cdot x^m\},$$

[1] The singular value decomposition is computed with the *Eigen* template library, accessible via `eigen.tuxfamily.org/` (lastly called up by the 6th May, 2016)

$$H_i^- = \{x \mid -U_{\bullet,i}^T \cdot x \leq -\min_{\overline{x} \in \overline{X}} \{U_{\bullet,i}^T \cdot \overline{x}\} - U_{\bullet,i}^T \cdot x^m\},$$

where $x \in \mathbb{R}^d$. All resulting normals of the ORH are thus simply defined by the corresponding non-inverted/inverted matrix columns of U and are always normalised. For the correct offset, the positive/negative product of the translated sample points and the normal needs to be maximised/minimised. The associated parts of the equation are very similar to the optimisation problem that a support function defines for the whole convex object (cf. Definition 3.13) with the significant difference, that the problem is in this case limited to a presumably much smaller and, more importantly, already available set of points from the object. My implementation therefore just iterates over all given sample points and determines the maximal/minimal product. The addition/substraction of the product of normal and arithmetic mean point is necessary in order to obtain the correct distances regarding the original sample points, i.e. these parts of the above equations reverse the initially conducted translation.

Example 4.1 (Computation of an ORH). *An oriented rectangular hull of the set of sample points* $X = \{(1,4)^T, (2,5)^T, (4,1)^T, (5,2)^T\}$ *is to be computed: The translated sample points are obtained by subtracting the arithmetic mean*

$$x^m = \frac{1}{4} \sum_{j=1}^{4} x^j = \begin{pmatrix} 3 \\ 3 \end{pmatrix}$$

from every sample point, yielding the set of translated samples $\overline{X} = \{(-2,1)^T, (-1,2)^T, (1,-2)^T, (2,-1)^T\}$. *For the sample matrix* S *thus holds*

$$S = \begin{pmatrix} -2 & -1 & 1 & 2 \\ 1 & 2 & -2 & -1 \end{pmatrix},$$

and the sample covariance matrix $Cov(S)$ *is obtained by*

$$Cov(S) = \frac{1}{3} \cdot \begin{pmatrix} -2 & -1 & 1 & 2 \\ 1 & 2 & -2 & -1 \end{pmatrix} \cdot \begin{pmatrix} -2 & 1 \\ -1 & 2 \\ 1 & -2 \\ 2 & -1 \end{pmatrix} \approx \begin{pmatrix} 3.33 & -2.66 \\ -2.66 & 3.33 \end{pmatrix}.$$

The singular value decomposition is in this case unique and given with

$$Cov(S) \approx \underbrace{\begin{pmatrix} -0.7 & 0.7 \\ 0.7 & 0.7 \end{pmatrix}}_{U} \cdot \underbrace{\begin{pmatrix} 6 & 0 \\ 0 & 0.667 \end{pmatrix}}_{\Sigma} \cdot \underbrace{\begin{pmatrix} -0.7 & 0.7 \\ 0.7 & 0.7 \end{pmatrix}}_{V^T}.$$

At last, four half-spaces are extracted from the matrix U. *These four half-spaces* H_1^+, H_1^-, H_2^+ *and* H_2^- *are defined by the sets*

$$\begin{aligned} H_1^+ &\approx \{x \mid (-0.7, 0.7) \cdot x \leq 2.1 + 0\}, \\ H_1^- &\approx \{x \mid (0.7, -0.7) \cdot x \leq -2.1 - 0\}, \\ H_2^+ &\approx \{x \mid (0.7, 0.7) \cdot x \leq 0.7 + 4.2\}, \\ H_2^- &\approx \{x \mid (-0.7, -0.7) \cdot x \leq 0.7 - 4.2\}, \end{aligned}$$

with the addition/substraction representing the reversion of the translations. The whole exemplary computation is illustrated by Figure 4.1.

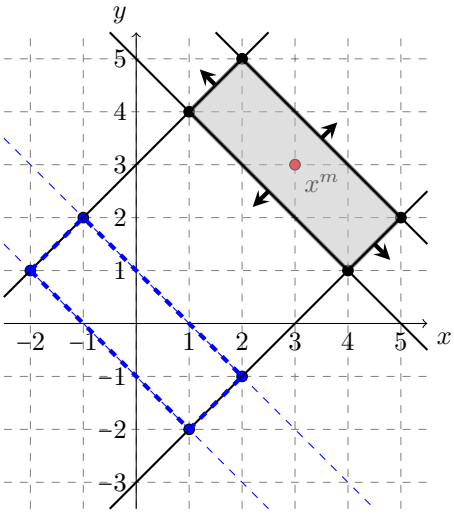


Figure 4.1: The resulting oriented rectangular hull with sample points and normals (black), and the implicitly computed translated ORH that is centered around the origin with translated samples and the two deviating half-spaces (blue). The red dot represents the arithmetic mean x^m of the sample points.

Oriented rectangular hulls have multiple applications regarding not only the conversion of state set representations: The naive practice is to use the vertices of an object as the underlying sample points for conversion purposes which is already a handy usage taken by itself, but there are even more possibilities, some of which will be explored throughout the rest of this chapter and Chapter 5.

4.2 Conversion of V-Polytopes

The first group of my conversion algorithms is dealing the transformation of V-polyhedra: With the finite set of vertices $V = \{v_1, ..., v_n\}$ at hand, conversion revolves around iterating over V in one or another fashion, be it in context of the already presented oriented rectangular hulls or other approaches. As already mentioned before, there exist no unbounded polytopes when using the definition introduced in Section 3.1 which is why I assume that all V-polyhedra are constructed according to that definition (and are thus V-polytopes). Since the state sets encountered during reachability analysis are usually bounded, this restriction is not of high relevance.

4.2.1 V-Polytope ↣ H-Polytope

Exact conversion from V-representation to H-representation is realised with a variation of the *QuickHull* algorithm (e.g. [Edd77]), an algorithm mainly designed for computing the convex hull of a set of points. This set of points is in our case of course the set of vertices V (possibly containing redundant vertices). Since this algorithm was not implemented by myself, I will only shortly describe the underlying procedure in the following with a visual example provided by Figure 4.2:

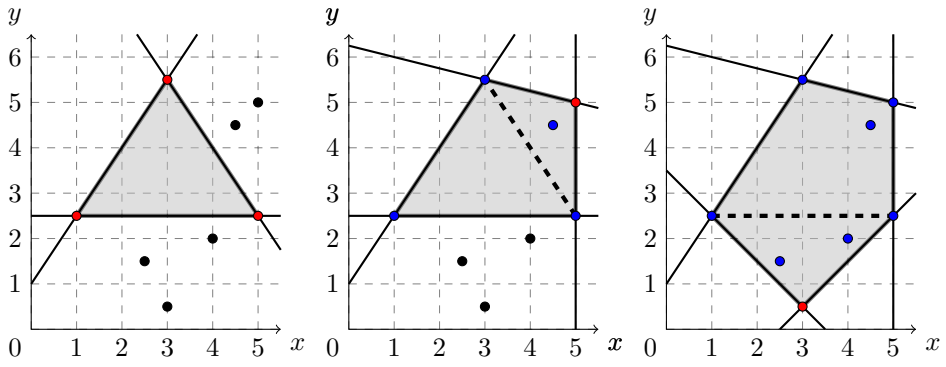


Figure 4.2: Paradigmatic three-step computation run of the described QuickHull variant: The algorithm first constructs a simplex, i.e. a triangle in the 2-dimensional Euclidian space (left). The full convex hull in H-representation is constructed via two further iterations (middle and right). Vertices that are chosen in the current step are displayed in red, possible vertex candidates for the next iteration are visualised in black, and points that are no longer considered are represented in blue. The dashed lines stand for the facets that are discarded in the corresponding iteration.

The iterative version of QuickHull that HyPro uses, starts by constructing a *simplex*[1] out of $d+1$ vertices from V with preferably large distance between each of them. Construction of a simplex means in this context that the half-spaces, whose intersection defines the above-mentioned simplex, are calculated; this is easily achievable for such a simple object.

If all other vertices, aside from the $d+1$ necessarily included points, are already lying inside the simplex, the algorithm terminates; otherwise, it carries on by extending the simplex and deleting redundant half-spaces in the process. This is done as described below:

For every extension step, a new vertex v_{new} lying at the outside of the object is chosen in order to extend the object by this vertex; this implies, that after every extension step (and also after the initial computation of the simplex), all points that are at that moment lying at the inside of the object are not further considered for the computation. Regarding the chosen vertex for each iteration, a vertex that is as far away as possible from the current object is preferable.

The extension itself is attained by calculating the *horizon* Ξ of the newly chosen vertex v_{new}, i.e. the set of vertices and half-spaces that are "visible" from the position of v_{new}: New connecting half-spaces between v_{new} and the visible vertices are calculated and added to the set of existing ones, while discarding the visible half-spaces, as these are assumed to be superfluous.

The described extension procedure is repeated until all vertices are included in the resulting object; the output object is then an exact H-representation of the source polytope.

[1] A simplex is the simplest full-dimensional polyhedron, i.e. an object with exactly $d+1$ vertices.

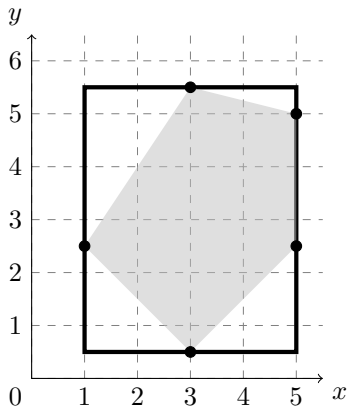


Figure 4.3: The V-polytope VP (gray area), its vertices, and the resulting box B.

4.2.2 V-Polytope ↠ H-Polytope

Since the exact conversion from V-representation to H-representation, described in the previous section, has exponential complexity in general, I present an alternative conversion for this purpose that over-approximates the resulting H-representation using oriented rectangular hulls (cf. Section 4.1): The ORH-algorithm is simply called with the set of vertices V as input; because the resulting ORH fully encloses V, it is not possible that the computed hull is smaller than the original V-polytope.

This approach shows that ORHs can not only be used for the conversion of convex objects, but also implicitly for the reduction of polyhedra that grow too large in terms of representation size as well: As a reminder, the resulting H-representation features only $2d$ half-spaces and when converted back to a V-polytope in an exact fashion, 2^d vertices.

4.2.3 V-Polytope ↠ Hyperrectangle

Transforming a Polytope into a box usually demands for an over-approximation due to the huge difference in expressiveness of the representation. The concept for the conversion of a V-polytope revolves around iterating over the set of vertices V once and determining the maximal and minimal component values for every dimension by memorisation. These extreme values, when used as the lower/upper interval bounds, implicitly define the intervals of the interval hull.

Example 4.2. *A V-polytope VP with $V = \{(1, 2.5)^T, (3, 0.5)^T, (3, 5.5)^T, (5, 2.5)^T, (5, 5)^T\}$ is converted to a hyperrectangle using the above-described approach: The procedure iterates over the given set of vertices and identifies the extreme values per dimension, yielding the box and interval hull $B = [1, 5] \times [0.5, 5.5]$ (cf. Figure 4.3).*

4.2.4 V-Polytope ↠ Zonotope

Considering that oriented rectangular hulls are special zonotopes and need to be over-approximated in the general case when having a polytope as the source object, I make use of the ORH-computation here as well, employing the vertices V as the input for

the program. Albeit, as the ORH-procedure (cf. Section 4.1) constructs the ORH in the form of half-spaces, there are additional computations required in order to calculate the centre c and the set of generators $G = \{g_1, ..., g_n\}$ of a proper zonotope:

Calculation of the centre c is realised by computing the V-representation of the obtained ORH in an exact manner (cf. Subsection 4.3.1) first and then determining the arithmetic mean of the vertices which is exactly the centroid of an object with such predefined simple geometry.

However, as already mentioned before, this is in general not the case when considering more complex objects; for such geometry, the arithmetic mean merely depicts an approximation of the centroid. Because of this, I had to discard my initial idea of reusing the arithmetic mean point x^m that is computed during the ORH-procedure so as to save computation time: The mean of the input sample points is not necessarily the centroid of the resulting ORH. Computation of a new centre point is thus necessary.

It is worth noting that there is no risk of eventual redundant vertices possibly disturbing the computation of the centroid: The ORH constitutes a minimal half-space representation of the over-approximation; this approximation is then converted exactly (the program would even eliminate superfluous points). As there are only 2^d vertices in the resulting V-representation, none of these vertices can be redundant.

The generators are, after the calculation of c, obtained in two phases:

1. Primarily, the objective is to obtain the point-to-plane-distances between the centre c and the corresponding bounding hyperplanes of the ORH-half-spaces. There are d such distances to compute, as the distance from c to both hyperplanes, regarding every output pair of the ORH-procedure, is exactly the same. In order to do this, arbitrary points p_i that lie on one of the two planes concerning every pairing are calculated beforehand: Every of the d points is simply obtained by computing the intersection point of one of the corresponding hyperplanes with one axis. With those points p_i available, the point-to-plane-distance ζ_i between centre c and the corresponding pair of bounding hyperplanes regarding the half-spaces H_i^+, H_i^- with normals $\vec{n}$ and $-\vec{n}$ is given by

$$\zeta_i = \frac{\vec{n}^T \cdot c - \vec{n}^T \cdot p_i}{\|\vec{n}\|^1}.$$

2. Secondly, the normals of the half-spaces need to be scaled to generator length. This is done by using the computed point-to-plane-distances: When being confronted with already normalised normals (which is the case when using normals from ORHs), the correct scaling factor α_i for each generator g_i is simply the distance ζ_i. If this is not the case, the scaling factors α_i would instead be obtained via dividing the distance by the length of the normals:

$$\alpha_i = \frac{\zeta_i}{\|\vec{n}\|}$$

The same reasons that prevent me from reusing the arithmetic mean of the sample point cloud x^m as the centre c of the resulting zonotope, prohibit that the offsets of the translated sample points, which are implicitly computed in the context of the ORH-procedure, can be utilised for obtaining the correct distances to c: Distances from the origin to the hyperplanes of the translated ORH, located at the origin, can

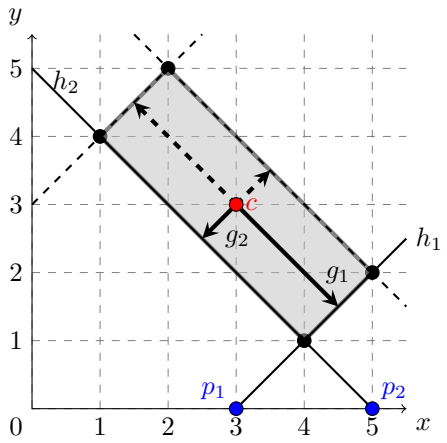


Figure 4.4: The ORH in G-representation with the vertices that were used for the mean computation (black points), the centre c (red), the plane points p_1 and p_2 (blue), and the two hyperplanes h_1 and h_2 that were used for the distance computation. The inverse of the generators g_1 and g_2 are represented by dashed arrows.

indeed differ from the correct distances concerning the centroid of the ORH and make the above-described steps necessary.

Example 4.3. *So as to clarify the derivation of a suitable zonotope, the oriented rectangular hull that was obtained in Example 4.1 is now transformed into G-representation (cf. Figure 4.4:*

A conversion to V-representation yields the set of vertices $V = \{(1,4)^T, (2,5)^T, (4,1)^T, (5,2)^T\}$ *which coincides in this case with the set of input sample points for the ORH-procedure. The centre of the zonotope is therefore given as the arithmetic mean of the vertices which results in* $c = (3,3)^T$.

The plane points p_1 *and* p_2 *are defined by the intersections of the x-axis and hyperplanes* h_1 *and* h_2 *for this example*[1] *, resulting in* $p_1 = (3,0)$ *and* $p_2 = (5,0)$. *For the distances* ζ_i *thus holds*

$$\zeta_1 \approx \frac{(0.7,-0.7)^T \cdot \begin{pmatrix}3\\3\end{pmatrix} - (0.7,-0.7)^T \cdot \begin{pmatrix}3\\0\end{pmatrix}}{\underbrace{\sqrt{0.7^2 + -0.7^2}}_{\approx 1}} \approx -2.1$$

and

$$\zeta_2 \approx \frac{(-0.7,-0.7)^T \cdot \begin{pmatrix}3\\3\end{pmatrix} - (-0.7,-0.7)^T \cdot \begin{pmatrix}5\\0\end{pmatrix}}{\underbrace{\sqrt{-0.7^2 + -0.7^2}}_{\approx 1}} \approx 0.7.$$

As the normals are unit vectors, the correct scaling factors α_i *are precisely the corresponding distances, yielding the set of generators* $G = \{g_1, g_2\}$ *with* $g_1 = (-1.47, 1.47)^T$

[1] If there had not been a possible intersection with the x-axis, i.e. the x-component of the normal had been zero, the algorithm would have chosen a different axis.

and $g_2 = (-0.49, -0.49)^T$.

By now, it should have become apparent that converting to a zonotope when using oriented rectangular hulls is more costly than sticking with the H-representation. It is thus recommended to use over-approximations from V-polytopes to H-polytopes, unless a zonotope is explicitly required.

4.2.5 V-Polytope ↠ Support Function

The transformation to a support function is in the context of HyPro, as already mentioned before, in theory no different from converting exactly to an H-polyhedron. Thus, the conversion in this context works exactly as described in Subsection 4.2.1, with the single difference of saving the result internally as a diverse datastructure that provides distinct functionalities.

4.3 Conversion of H-Polytopes

When the source object is a polytope in H-representation, the naive approach in terms of conversion revolves around initially determining the vertices of the underlying object; this describes the concept of nearly all of the developed algorithms in this subsection in fact very well.

Although it is easily possible to construct unbounded H-polyhedra, I assume here, similarly to the situation with V-polyhedra before, that the input objects are all bounded, i.e. that they are all H-polytopes; unbounded objects are, as already mentioned, of low interest for reachability analysis.

As a reminder, the notation H_i represents the half-spaces of the H-polytope with the integer n denoting the number of half-spaces in this context.

4.3.1 H-Polytope ↠ V-Polytope

Obtaining the set of vertices of geometry that is represented by an intersection of finitely many half-spaces comes down to intersecting the space-dividing hyperplanes of all given half-spaces until all extreme points are found:

Regarding the d-dimensional Euclidian space, any intersection concerning the bounding hyperplanes of d different half-spaces may yield a full-dimensional intersection point in case the normals of the hyperplanes are affinely independent. These intersection points include the extreme points of the polytope and therefore all vertices necessary for a minimal representation. With this coherence follows that, when converting from H-representation to V-representation, each of those intersection points is a possible vertex candidate for the transformation result.

An appropriate exact conversion procedure thus builds all possible d-combinations of given half-spaces and computes the intersection point of the corresponding hyperplanes for every combination (if there is one). Each resulting point then has to be examined further: Only if the vertex candidate fulfills all constraints of the remaining half-spaces, it is truly a vertex of the corresponding V-representation; all candidates are therefore tested against each of the $n - d$ remaining half-space inequations.

Figure 4.5 illustrates the above-described concept in form of a visual example.

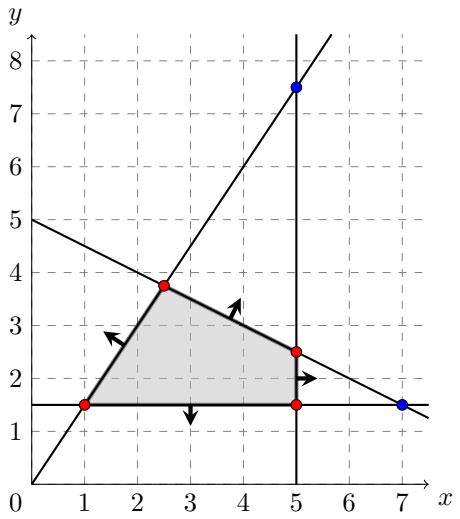


Figure 4.5: A 2-dimensional polytope P (gray area) that is transformed exactly from H-representation to V-representation: Every possible pairing of the displayed half-spaces yields a vertex candidate by intersection of the corresponding hyperplanes. These resulting six vertex contenders are tested for membership of all half-spaces; two candidates are eventually discarded (blue points), and the other four candidates form the set of vertices concerning P (red points).

4.3.2 H-Polytope ↠ Hyperrectangle

Out of the two transformation approaches that I developed for the over-approximative conversion from H-polytope to hyperrectangle, the procedure of this subsection represents the naive algorithm: The H-representation is initially transformed into a vertex-representation as described in the previous subsection, before conducting an over-approximative conversion to a hyperrectangle according to Subsection 4.2.3.

4.3.3 H-Polytope ↠ Hyperrectangle (Alternative Approach)

With the ambition of avoiding the costly computation of the vertices, I had the idea of using a linear optimiser for obtaining an over-approximating hyperrectangle, exactly the way it is utilised for evaluating a support function. Since the underlying geometry of HyPro support functions is mostly represented by H-polyhedra, this alternative approach for transforming H-representations into boxes works just like transforming support functions into hyperrectangles which is presented in Subsection 4.6.5, alongside an example.[1]

Both the naive conversion procedure from H-representation to box and the algorithm using linear optimisation for the same purpose are contrasted in terms of efficiency in Subsection 5.1.2.

[1] Please note that I do not present the corresponding procedure in this section, as it is more fittingly to discuss this algorithm in context with the algorithms for the transformation of support functions.

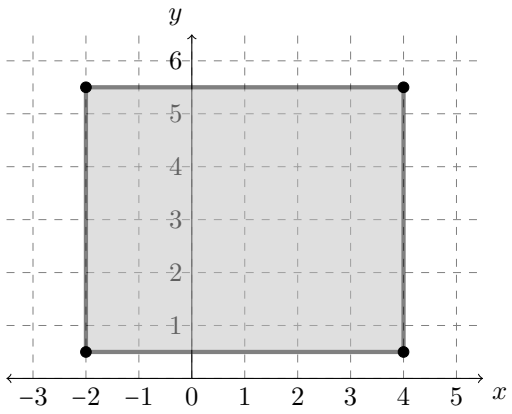


Figure 4.6: The box B, converted into a V-polytope.

4.3.4 H-Polytope ↠ Zonotope

Regarding the conversion to zonotopes, the only available approach in this scenario is an initial conversion to V-representation before continuing with an over-approximation to a resulting zonotope (cf. Subsection 4.2.4).

4.3.5 H-Polytope ↣ Support Function

There is only one necessary measure for the transformation of an H-polytope into a support function, and that is a simple constructor call in order to define the source H-polytope as the underlying geometry of a new support function.

4.4 Conversion of Hyperrectangles

The conversion of hyperrectangles into other representations is, as one might expect, a very simple task. With a constant number of vertices and facets and also fixed facet normals, all algorithms are very economical to perform, since they only depend on the dimension d. Furthermore, all of the necessary data is instantly available with the given defining intervals of the box. In the context of this thesis, I presume that the intervals are non-empty and bounded.

As boxes form the most limited representation type that is tackled here, at least in terms of precision, approximating the resulting geometry is not sensible; exact transformations are always possible and advised for hyperrectangles.

Regarding the currently treated hyperrectangle, the set of the d box intervals is denoted by $I = I_1 \times ... \times I_d$ with $I_i = [\underline{x_i}, \overline{x_i}]$.

4.4.1 Hyperrectangle ↣ V-Polytope

When transforming a box into a V-polytope instead, the 2^d resulting vertices are simply determined by building all possible combinations of d distinct interval bounds (of which there are exactly 2^d combinations); each combination yields one resulting vertex.

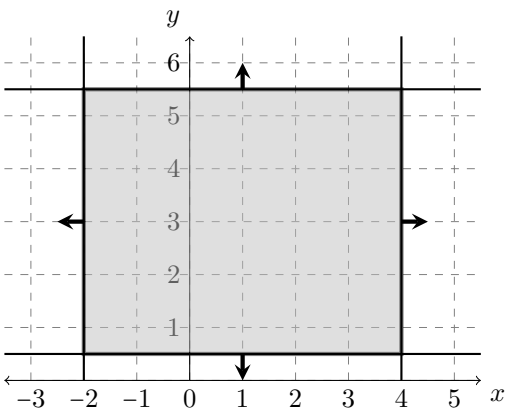


Figure 4.7: The box B as the intersection of the half-spaces resulting from the conversion to an H-polytope. Its facet normals are indicated by arrows.

Example 4.4. *Box* $B = [-2,4] \times [0.5,5.5]$ *is at last converted into a* V*-polytope which is illustrated by Figure 4.6. The resulting set* V *of vertices is given with all possible combinations of interval end points of* B *and is presented below:*

$$V = \{\begin{pmatrix}-2\\0.5\end{pmatrix}, \begin{pmatrix}-2\\5.5\end{pmatrix}, \begin{pmatrix}4\\0.5\end{pmatrix}, \begin{pmatrix}4\\5.5\end{pmatrix}\}$$

4.4.2 Hyperrectangle ↣ H-Polytope

When striving for an H-representation of a hyperrectangle, only the offsets of the $2d$ half-spaces can vary, the $2d$ normals are fixed in the form of the positive axes directions and the negative axes directions.

The offsets for every pair of half-spaces are received by taking the non-inverted upper interval bound $\overline{x_i}$ as the proper distance from the origin to the bounding hyperplane regarding the corresponding positive axis direction, and using the inverted lower interval bound $-\underline{x_i}$ for the negative axis direction analogously. This cohesion is easily deducible when rearranging the corresponding half-space inequalities.

Example 4.5. *The returning box* B $(B = [-2,4] \times [0.5,5.5])$ *is converted into an* H*-polytope* (C,z) *this time (cf. Figure 4.7): The normal matrix* C *is predefined with*

$$C = \begin{pmatrix}1 & 0\\-1 & 0\\0 & 1\\0 & -1\end{pmatrix}.$$

The proper distance vector z *is then given with the interval bounds (non-inverted upper bound and inverted lower bound):*

$$z = \begin{pmatrix}4\\2\\5.5\\-0.5\end{pmatrix}$$

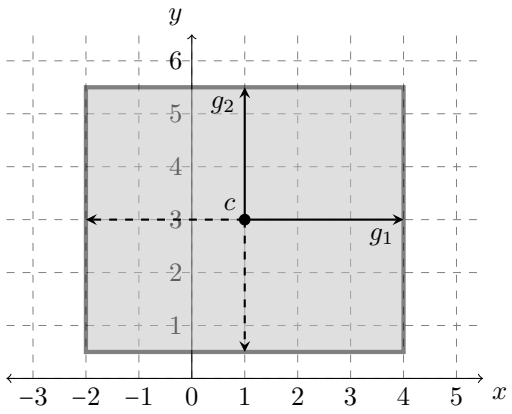


Figure 4.8: The box B with the resulting centre c and generators g_1, g_2 after the transformation. The inverse of the generators are indicated by dashed vectors.

4.4.3 Hyperrectangle ↣ Zonotope

In order to transform a box into a zonotope, the corresponding centre c and generators $G = g_1, ..., g_d$, of which there are d many for every hyperrectangle, are calculated as described below:

- The centre c is given implicitly by the centres of the intervals, i.e. the ith zonotope centre component c_i is given by
$$c_i = \frac{\underline{x_i} + \overline{x_i}}{2}.$$
- Concerning the generators g_i, their alignment is fixed (every generator points in one axis direction), and thus only their independent lengths ℓ_i have to be computed, which is achieved by halving the length of the associated interval (also called the *radius* of the interval):
$$\ell_i = \frac{|\underline{x_i} - \overline{x_i}|}{2}$$
The resulting generator is then given via $g_i = (0, ..., \ell_i, ..., 0)^T$ with ℓ_i residing at the ith position in the vector.

Example 4.6. *A box B, defined by $B = [-2, 4] \times [0.5, 5.5]$, is to be converted to a zonotope (cf. Figure 4.8): The centre c is computed via*
$$c = \begin{pmatrix} \frac{-2+4}{2} \\ \frac{0.5+5.5}{2} \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \end{pmatrix},$$
the generators $G = \{g_1, g_2\}$ are calculated with
$$g_1 = \begin{pmatrix} \frac{|-2-4|}{2} \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ 0 \end{pmatrix}$$
and
$$g_2 = \begin{pmatrix} 0 \\ \frac{|0.5-5.5|}{2} \end{pmatrix} = \begin{pmatrix} 0 \\ 2.5 \end{pmatrix}.$$

4.4.4 Hyperrectangle ↣ Support Function

As one would expect, the underlying procedure in this constellation resembles an exact conversion from hyperrectangle to H-polytope (cf. Subsection 4.4.2).

4.5 Conversion of Zonotopes

Regarding zonotopes, all realised procedures revolve around computing the vertices of the object first which is why the exact conversion from zonotope to V-polytope is the main focus of this section. This task of computing the extreme points of a zonotope is also known as the *zonotope construction problem*[Fuk04]. Similar to the other sections before, I only consider bounded source objects; unbounded zonotopes are undefined one way or the other.

As a reminder, c denotes the centre of a zonotope Z, while $G = \{g_1,...,g_n\}$ is the set of its generators.

4.5.1 Zonotope ↣ V-Polytope

Before dealing with the zonotope construction algorithm, I point a special property of zonotopes out:

A point v is only a possible extreme point of the zonotope, if it is the result of n subsequent additions of line segments, with the n summands being strictly the n generators of the zonotope; every generator represents exactly one summand, either in its "normal" form or as its inverse. This yields a total of 2^n vertex candidates for every zonotope.

With that said, in terms of HyPro, the zonotope construction problem is solved in a recursive manner: Starting at the position of the centre c, the algorithm obtains two positions p_1 and p_2, one by adding the first generator g_1 to c, and the other by adding $-g_1$ to c. If there is more than one generator, the procedure is then recursively called for both p_1 and p_2 (becoming the centre points of their recursion calls). The first generator g_1 is removed beforehand, such that g_2 is then the new first generator.

This course of action is repeated until the lowest recursion layer is reached, i.e. until there is only one generator left. The additions at this lowest layer then provide all 2^n extreme point candidates that are afterwards returned to the invoking instance of the procedure. Every recursion layer is therefore tasked with the addition of one generator and the corresponding inverse exclusively.

Although there are redundant vertices, lying inside of the object, among the 2^n calculated extreme point candidates, the algorithm does not identify these redundancies, as superfluous vertices are removed by the V-polytope constructor that is called subsequently.

Figure 4.9 illustrates the above-described approach.

4.5.2 Zonotope ↣/↠ H-Polytope

Constructions of H-polytopes out of zonotopes is no easy feat: Getting valuable information about the enclosing half-spaces efficiently with only the centre and the generators given has proven to be a difficult task for me, in fact, Fukuda states that computing an H-representation "is still an open question"(cf. [Fuk], p. 3). Because of

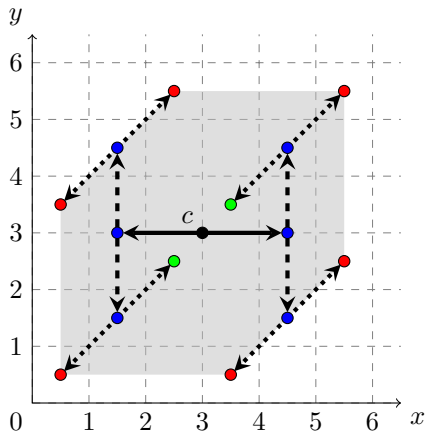


Figure 4.9: Paradigmatic zonotope construction for a zonotope Z with three generators. Line segments that correspond to the first generator and thus highest recursion layer are displayed by plain arrows, the additions conducted by the second layer are depicted by dashed arrows, and the computations of the final deepest layer are visualised by dotted arrows. Additional displayed objects are the centre point c (black), the points that are computed by the algorithm for further recursions, but are never returned, as they are no vertex candidates (blue), and the vertex candidates that are returned by the algorithm (red and green). The two green vertices are redundant and are removed by the V-polytope constructor subsequently.

this, when aiming for an H-representation, a detour via V-representation is currently common.

This means that an exact conversion from zonotope to V-polytope, as described in the previous subsection, is performed at first, before conducting either an exact conversion (cf. Subsection 4.2.1) or an over-approximative conversion, as described in Subsection 4.2.2, in order to obtain an H-representation of the object.

The choice of method is determined by passing one function parameter to the function and is thus up to the user to decide. The default conversion mode is an exact conversion, because it is conventional to use simple zonotopes (like oriented rectangular hulls) for reachability analysis. Regarding such basic zonotopes, exact convex hull algorithms performed on the set of vertices usually have an acceptable runtime in most applications.

4.5.3 Zonotope ↠ Hyperrectangle

Over-approximation of the in general much simpler shaped hyperrectangles, when being confronted with a zonotope as the source object, is yet again achieved by determining the vertices of the zonotope. After this exact conversion to a V-polytope (cf. Subsection 4.5.1), the resulting box is approximated as described in Subsection 4.2.3.

4.5.4 Zonotope ↠/↠ Support Function

Transformation to a support function when dealing with zonotopes is, as one might expect, done like a conversion to an H-polytope as well (cf. Subsection 4.5.2). Choosing a conversion mode is again up to the user, with exact conversion being the default mode.

4.6 Conversion of Support Functions

Last but not least, conversions of support functions are very different from any sort of conversion algorithm that was presented until here, and they come with very specific challenges: As already mentioned before, there is only one way of obtaining any information about the underlying geometry of convex sets that are represented by support functions, and this one way is evaluating the support function $p_\Omega(\ell)$ into any direction ℓ. When facing bounded objects, which is yet again assumed in this context, each evaluation of a support function yields a supporting hyperplane (cf. Chapter 3.4) that can then be used for an approximation of the underlying convex set Ω, as described in the following sections in more detail.

In general, the quality of a result in terms of precision is extremely dependent on a number of factors: The most important factors are the number of directions that the support function is evaluated into, and the nature of the underlying geometrical object: As a general rule holds that, the more directions the support function is evaluated into, the more precise is the resulting approximation and the more structurally complex the convex set, the more directions are required for an accurate approximation of the source object. Choosing sensible directions that fit to the geometry is also very helpful efficiency-wise, as a few well-chosen directions possibly obtain an approximation of similar or even better quality than a lot more randomly picked or uniformly distributed directions.

However, it is to be kept in mind that any evaluation results in the need for the solving of a whole linear optimisation problem; it is thus, in terms of computation time, not advised to evaluate into more directions than necessary. Assessment of how many and which directions are appropriate though is a huge challenge, since determining any information about the geometry beforehand is impossible with only the support function at hand, which serves as a kind of black box.

Use case scenarios that allow for an estimation of the underlying structure, concerning the support functions that are utilised, could heavily improve the choice of both number and alignment of directions and thus also computation time and quality of the approximation. My situation in the context of this bachelor thesis is, however, detached of any specific reachability analysis use case scenario, which is why I decided to use a variable number of uniformly distributed template directions for most of the algorithms in this chapter.

These template directions are computed by a special utility function which is called with a parameter u: The function begins by constructing a template of u uniformly distributed directions in the 2-dimensional plane. If $d > 2$ holds, this 2-dimensional template plane is then rotated and placed axis-aligned concerning every possible remaining pairing of dimensions, yielding more directions for higher dimensions with the same parameter u. Calling this function in the 3-dimensional Euclidian space with

u thus results in approximately $3u$ directions,[1] as the template is besides the initial xy-alignment further placed xz-aligned and yz-aligned. It is therefore not possible to construct every number of directions (for $d > 2$) when using this function.

Apart from that restriction, it is up to the user to decide how many template directions are being utilised in the corresponding conversion procedures. If nothing is specified, the default value is $u = 8$, as eight template directions per dimension pair provided good precision in most conducted tests.

It is also worth noting that support functions only very rarely obtain exact results when converted into other representations, especially when dealing with structurally complex objects. When gradually increasing the number of evaluation directions and keeping the previously used ones, a fixed-point will eventually be reached: No matter how complex the object, the output object will be an exact transformation of the orginal object at some point; further increase of the number of evaluation directions beyond this point has no other effect than a mounting of computation costs.

In the following, let $p_\Omega(\ell)$ be the support function of the convex set Ω, where ℓ is some d-dimensional direction vector. The integer m denotes the number of directions that the support function is evaluated into.

Before the presentation of my developed algorithms regarding the conversion of support functions, I take a closer look at a specific problem regarding support functions in the following subsection, since my inability to solve this issue prevents two of the implemented algorithms from working in all scenarios. The corresponding problem revolves around the computation of boundary points of objects that are represented by support functions.

4.6.1 Boundary Point Computation for Support Functions

Some of the conversion algorithms that I talk about in this section need (any) boundary points of the underlying geometry for their computations. Calculations of such boundary points have proven to be somewhat problematic: When evaluating a support function $p_\Omega(\ell)$ into any direction ℓ and receiving a supporting hyperplane H_ℓ in the process, it is certain that there is at least one boundary point lying somewhere on this plane, but it is unclear where exactly this boundary point is located.

In an attempt to solve this problem, the linear optimiser that is used by HyPro for the evaluation of support functions was modified: The last point p_{last} that is visited during a computation of the linear optimiser is now memorised and this very point is in addition to the evaluation direction ℓ forwarded through the whole sequence of operations that were conducted on the support function (cf . Subsection 3.6.1); this is achieved by applying all memorised operations to p_{last} one after another. This last visited point p_{last} is in favourable scenarios a boundary point of the underlying geometry, and is in fact (nearly) always an extreme point of at least one underlying object[2]. When there are binary operations involved, the optimiser obtains one such point p_{last} for every object that was used as an operand concerning the memorised operations.

While this modification means that every conducted evaluation has now increased cost in comparison to before, computing those operations for a few points in addition

[1] Depending on the value of u, there may be duplicate directions among the $3u$ initially computed directions that are removed afterwards.

[2] The only scenario in which p_{last} is not necessarily an extreme point, is a situation in which the support function is exaclty evaluated in direction of a facet normal.

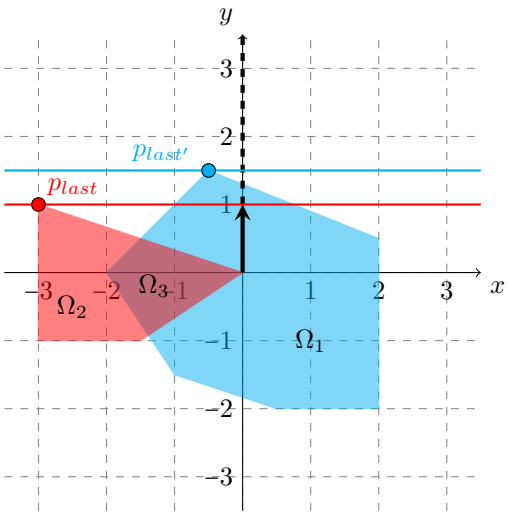


Figure 4.10: Two convex sets Ω_1 and Ω_2, both represented by a support function, are intersected with each other, resulting in the convex set Ω_3. The set Ω_3 shall now be evaluated into direction $\ell = (0,1)^T$ for a boundary point: In order to do this, the linear optimiser evaluates in both Ω_1 and Ω_2 once and returns in addition to the offset of the supporting hyperplanes (which are displayed in the colour of their corresponding set) also the lastly visited point (p_{last} and $p_{last'}$). Afterwards, the minimal point out of these two points regarding direction ℓ is chosen as a boundary point which is in this case p_{last} (the red point). It is obvious that p_{last} is no point of the actual intersection set Ω_3 and thus also no viable boundary point of Ω_3.

is not very expensive when limiting the memorised operations to a maximum number, and is still in every scenario more efficient than computing the same operations for the whole geometry.

However, this solution attempt is flawed, and this flaw is substantiated by the current realisation of the intersection operation concerning support functions of the HyPro project: As already described in Subsection 3.6.1, when intersecting a support function with any other object, the resulting intersection set is implicitly over-approximated by evaluating into some direction ℓ in both underlying objects and taking the minimum of both result values. This lack regarding an exact computation of the intersection makes it impossible to calculate valid boundary points using the above-described idea. The whole problem becomes a lot clearer with the visual example that is provided by Figure 4.10.

The situation concerning the boundary point computation of a support function that memorised an intersection operation becomes even worse when the result of that intersection is empty: Although the boundary is non-existent for an empty intersection, the boundary point computation still returns result points. These result points are obviously wrong, but are never identified as incorrect results, since empty intersections currently cannot be detected.

With no idea for a possible new approach for the intersection operation when dealing with support functions, additional effort was put into the development of different boundary point computation procedures, but without success. Albeit Pijush

K. Ghosh and K. Vinod Kumar presented a theoretical way of obtaining the boundary of a support function in [GK98], it is unclear how this approach should be realised without any intersection operation:

In short, when evaluating in direction ℓ, Ghosh and Kumar state that, concerning the support function, the computation of the *directional derivative* at ℓ in direction of unit vectors w in all directions yields a complete facet of the object with a dimension of at most $d-1$. Repeating this technique d times thus necessarily results in an object of dimension 0, i.e. a point. This point is then a boundary point of the underlying geometry.

However, it is unclear how this idea should be realised in praxis, with one of the main issues being the determination of a suitable algebraical representation of the support function (which is obviously needed for the calculation of the directional derivatives), at least without restricting the underlying geometry to only allow simple objects. Another problem is that there is an unlimited number of unit vectors which would make an approximation of the resulting facets necessary.

As a workaround to these problems, I started developing an implementation that would make the currently realised boundary point computation via the modification of the linear optimiser dispensable. I could think of only one possible other way of calculating the facet of an object represented by a support function, which is to intersect the support function with the supporting hyperplane that is obtained by evaluating in some direction ℓ. But as should have become clear by now, computation of the intersection does not work properly at the moment and it was not possible to resolve the corresponding problems, which is why this unfinished implementation was eventually discarded.

Concerning all following conversion algorithms, boundary point computation is thus achieved with the modified linear optimiser described above. Circumventing its issues without sacrificing the intersection operation, which is very frequently used in reachability analysis, is possible in the following way: By converting the support function to a different representation before performing an intersection and transforming the result back to a support function after the conducted intersection operation, the correctness of future operations is ensured and the list of memorised operations is reset as a positive side effect.

4.6.2 Support Function ↠ H-Polytope

As already mentioned, evaluating a support function $p_\Omega(\ell)$ into any direction ℓ yields a supporting hyperplane. The idea for my conversion to an H-polytope is to evaluate $p_\Omega(\ell)$ into m template directions and to use each of the resulting supporting hyperplanes as a bounding hyperplane of the corresponding half-space. The output object is then defined by the intersection of those m resulting half-spaces.

This procedure always results in an over-approximating H-polytope because of one particular interesting property of supporting hyperplanes: Supporting hyperplanes only touch the underlying geometry and never intersect with the inner points; there is thus no danger of reducing the original object, when relying on such planes. This coherence should become clear with Figure 4.11.

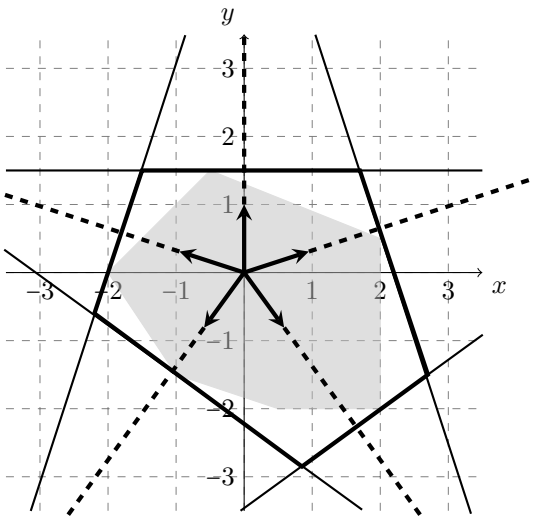


Figure 4.11: A convex set Ω, depicted by the gray area, is evaluated into five uniformly distributed template directions. The directions are represented by arrows, with the dotted lines denoting the extension of these directional vectors. The resulting five supporting hyperplanes yield a visualised over-approximation of Ω; the facets of the approximation are displayed via a thick outline.

4.6.3 Support Function ↠ V-Polytope

The attendant approach for obtaining a suitable over-approximation in V-representation is working exactly as the procedure described in the previous subsection, with the addition of an exact conversion to a V-polytope (cf. Subsection 4.3.1) after the construction of an over-approximating H-polytope based on the derived supporting hyperplanes.

A direct approach for the computation of proper vertices as a special case of the boundary point computation for support functions (cf. Subsection 4.6.1) is in theory possible, but is combined with a high computational effort and not very robust either: For a proper over-approximation, it must be ensured that at least all extreme points of the objects are obtained; this would require multiple over-approximations of the source object which would have to be compared in the process.

4.6.4 Support Function ⇢ V-Polytope

Under-approximative conversions from support functions to V-polytopes work, despite being simple in concept, only restrictedly: The idea here is to obtain a number of boundary points of the underlying convex set Ω by evaluating the corresponding support function into a number of template directions, with each evaluation yielding one boundary point; these points then form the set of vertices of the resulting under-approximation. As only points on the boundary of Ω are considered, it is obvious that the result is always a subset of the original set. A visual example is presented with Figure 4.12.

As already stated in Subsection 4.6.1, the computation of boundary points is

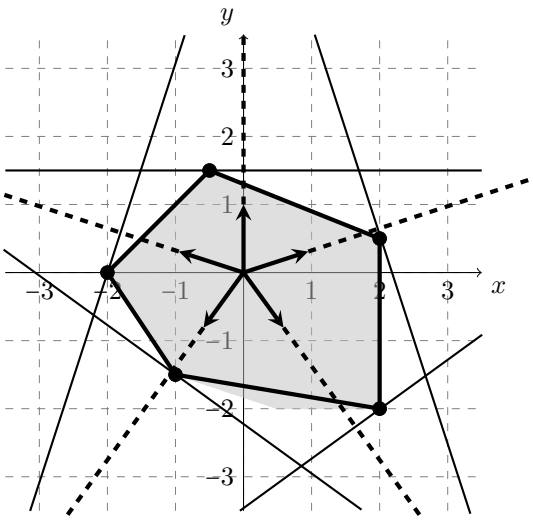


Figure 4.12: The support function of returning convex set Ω is now evaluated into the same five directions again, this time with the goal of obtaining an under-approximation. The resulting boundary points that are determined by the linear optimiser are displayed as black points and the resulting under-approximation is displayed with a thick outline. In this case, when using five template directions, the computed under-approximation is close to being the original set.

currently not working properly for support function objects that were intersected with other objects. The under-approximation of V-polytopes therefore in general only works correctly on input objects that feature no intersection as a memorised operation.

4.6.5 Support Function ↠ Hyperrectangle

My implementation of conversions from support functions to hyperrectangles is the only procedure in this section that does not rely on uniformly distributed template directions: For obtaining the interval hull of the underlying geometry, it is sufficient to evaluate the support function into the $2d$ directions of the axes and to use the resulting offsets as the corresponding interval bounds: Evaluation in negative axis direction yields a distance that is, when inverted, the correct lower interval bound of this dimension and evaluating in positive axis direction results in an offset that constitutes without further modifications already the proper upper interval bound (cf. Figure 4.13).

As already mentioned in Subsection 4.3.3, this technique is also used as an alternative approach for obtaining the interval hull of an H-polytope, since the linear optimiser works on H-polyhedra anyway for the most part.

4.6.6 Support Function ↠ Zonotope

The first developed approach for conversions to zonotopes, when being confronted with a support function as the source object, recombines already presented tech-

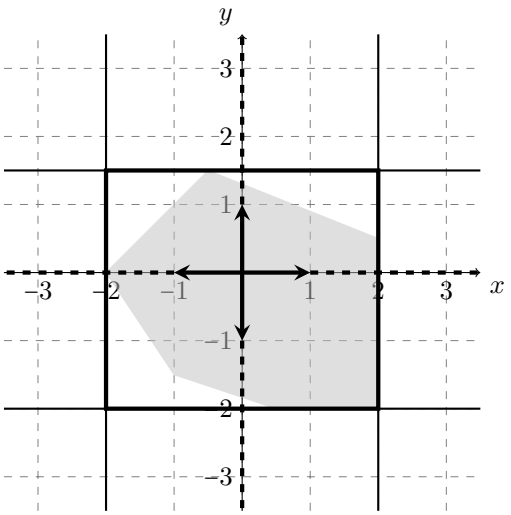


Figure 4.13: Convex set Ω (gray area), represented by a support function, is converted over-approximately to a box (visualised by a thick outline). This is achieved by evaluating in the $2d$ axis directions and deploying the obtained offsets as the corresponding interval bounds with the received offsets in negative axis directions being negated in the process.

niques: At first, the underlying geometry is converted to an H-polytope as described in Subsection 4.6.2. With the resulting half-space-representation that is in general no zonotope yet at hand, an oriented rectangular hull, with the vertices of the obtained H-polytope (received via an exact conversion to V-representation) as input, is computed (cf. Subsection 4.2.4).

4.6.7 Support Function ↠ Zonotope (Alternative Approach)

As the procedure described in the previous subsection features a lot of smaller conversions that are costly for complex objects and escpecially expensive in higher dimensions, I developed a different approach for the conversion to zonotopes with the ambition of improving computation time:

This alternative approach first computes a set of boundary points of the support function (cf. Subsection 4.6.1) with template directions, before performing the ORH-computation on just these boundary points, saving a lot of costly transformations thereby. Aside from the problem that the boundary point computation is not working properly with intersection, it is likely that the ORH obtained this way does not enclose the whole underlying geometry, meaning that there is additional effort needed in order to expand the calculated ORH such that it constitutes a proper over-approximation of the underlying object.

Thankfully, it is only necessary to conduct a fixed number of $2d$ additional evaluations afterwards, since we know that the ORH has only $2d$ facets with already known normals. The procedure thus evaluates in each of the $2d$ facet normal directions and the newly obtained offset then determines how far the facets of the ORH need to be pushed in the corresponding direction so as to it encloses the object in combina-

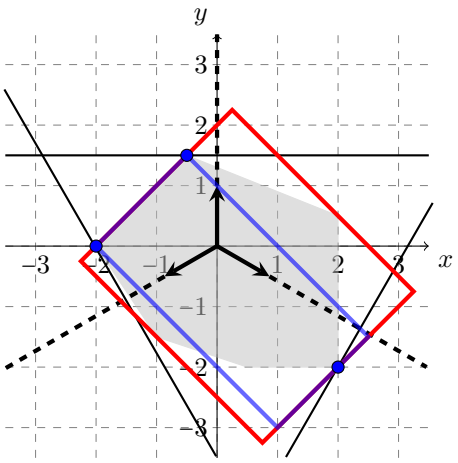


Figure 4.14: Underlying convex set Ω (depicted by gray area), represented via a support function, is now converted into a zonotope: Using three template directions, three corresponding boundary points (blue points) are obtained, and the initially computed ORH, based on those three boundary points is displayed via a blue outline. As this ORH is no proper over-approximation of Ω, all facets are pushed back by correct offset values that are determined by conducting additional evaluations into the $2d$ ORH facet normals. The output ORH is indicated by the red outline (ORH normals, centre, and generators are not visualised due to the already crowded graphic).

tion with the other facets. Figure 4.14 visualises a paradigmatic application of the algorithm.

While the procedure featured in this subsection works only restrictedly because of the boundary point computation, precision of the resulting approximations was assumed to be a lot better when developing this technique. Both approaches are compared in terms of precision and computation time in Subsection 5.2.2.

Chapter 5

Evaluation

With all of my conversion algorithms presented, I now take a closer look at the correctness of the approaches and also the precision and the computation speed that these procedures provide:

Section 5.1 describes a general verification of all of the introduced transformation approaches in context of a universal conversion scenario featuring special template objects, while Section 5.2 deals with two specific experiments that involve only one or two of the portrayed algorithms each.

In contrast to the previous chapter, all graphics in this chapter are actual output plots of my implemented functions. All experiments were conducted on the same computer with background tasks kept to a necessary minimum.

In the context of this chapter, I use the following abbreviations:

- V for V-polytope
- H for H-polytope
- B for box
- Z for zonotope
- SF for support function

5.1 General Analysis

In order to be able to provide a comparison of all twenty-four conversion algorithms, I made use of template objects, i.e. objects that are built with the help of the already described special utility function which computes a number of template directions based on a passed parameter u (cf. Section 4.6):

Template H-polytopes consist of one half-space for every obtained template direction: Each resulting template direction forms the normal of a half-space with a fixed offset of five. Template objects in other representations were received by conversion of the initially constructed template H-polytopes into the other representations according to Section 4.3. The conversions were done exactly for template V-polytopes and template support functions, and were performed over-approximately for template hyperrectangles, resulting in the interval hulls, and template zonotopes, resulting in oriented rectangular hulls.

All template objects therefore have a conformal structure: In the 2-dimensional space, the parameter u exactly describes the number of resulting facets of the object, regarding higher dimensions, there are approximately u facets per possible dimensional component pair. Although objects which are that uniform are not encountered very often in praxis, they still give a good indication about the correctness, computation time and precision that is to be expected from those algorithms.

The general analysis described in this section was conducted in two phases: The first phase aimed for the production of output plots regarding the conversion of template objects in the second dimension (the 2-dimensional Euclidian space was chosen for reasons of presentability), while the second phase compared runtimes concerning the transformation of 3-dimensional template objects. Both phases and the corresponding obtained results are discussed in the following two subsections.

5.1.1 General Analysis - Plots

Every 2-dimensional source template object concerning the preperation of the plots was constructed with parameter $u = 12$, i.e. the source object for the conversions is a regular twelve-sided polygon (*dodecagon*), except for template hyperrectangles and template zonotopes; these represent the interval hull and the ORH of the mentioned polygon respectively. Regarding every group of algorithms, the constructed template object was simply transformed with every available conversion procedure once and the resulting output object was plotted afterwards. Subsequently, I present the obtained plots along with the source object in five groups with exactly the same algorithm groupings and presentation order as in Chapter 4.

Please note that the purpose of the plots in this subsection is mainly to give an indication about how the results of the different algorithms per group look in relation to each other.

Conversion of Polytopes

Both the conversions of H-polyhedra and V-polyhedra obtain exactly the same output plots, as visualised by Figure 5.1.

The results are unsurprising here, with the exact conversions working correctly and the over-approximations to zonotope and H-polytope obtaining the same ORHs. Any other alignment of the ORHs would have been as fitting due to the uniform structure of the source template object, but, in this scenario, the ORH gives no benefit precision-wise in comparison to the computed interval hull. Both implementations of the conversion from H-polytope to hyperrectangle result in the same output object and there is thus unsurprisingly no difference in terms of precision.

Conversion of Hyperrectangles

As the utilised template hyperrectangles already represent the interval hull of the template dodecagon and boxes in general only feature exact conversions, there is no use in showing additional plots concerning this group of algorithms, the obtained objects concerning the box-conversion procedures all look like the red object in Figure 5.1. This is perfectly fine, as only the internal datastructure representation was changed during the transformation process.

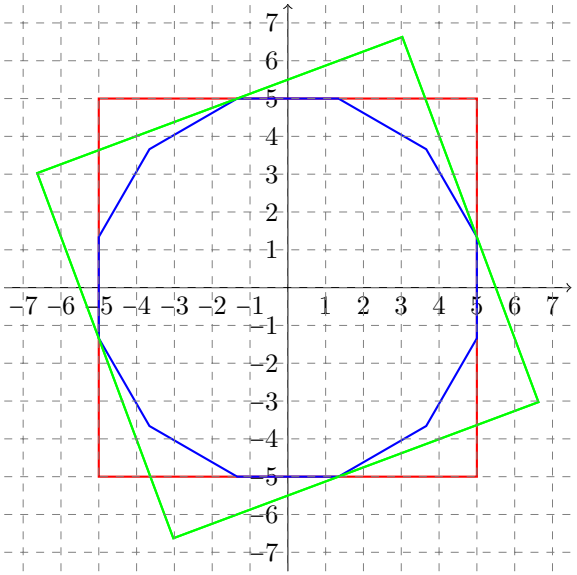


Figure 5.1: Output plots concerning the conversion of polytopes. The blue object is the source object and also the result of the conversions $V \nrightarrow H$,$V \nrightarrow SF$, $H \nrightarrow SF$, $H \nrightarrow V$; the green object depicts the ORH that was constructed by all of the transformations $V \twoheadrightarrow H$,$V \twoheadrightarrow Z$, $H \twoheadrightarrow Z$. Finally, the red outline represents the interval hull obtained by $V \twoheadrightarrow B$ and both versions of $H \twoheadrightarrow B$.

Conversion of Zonotopes

Since the template zonotopes are no different from the template boxes in this case, I constructed a non-template zonotope (with centre $c = (3,3)^T$ and generators $G = \{(0,1)^T, (1,1)^T, (2,-1)^T\}$) that was used as input for the corresponding conversion procedures instead. The results can be seen in Figure 5.2.

The output is satisfactory in this scenario as well, with the ORH providing a better approximation than the hyperrectangle this time.

Conversion of Support Functions

At last, a template support function of the dodecagon was converted into all other representations using the default eight template directions for evaluation, resulting in the output plots depicted by Figure 5.3.

All algorithms work correctly in this scenario as well and provide fairly good approximations: The obtained over-approximating polytope in both H-representation and V-representation is particularly tight, with the under-approximation in green being close to the source object, too. Regarding the two different approaches for conversion to zonotopes, the resulting ORHs provide the same precision although being aligned differently.

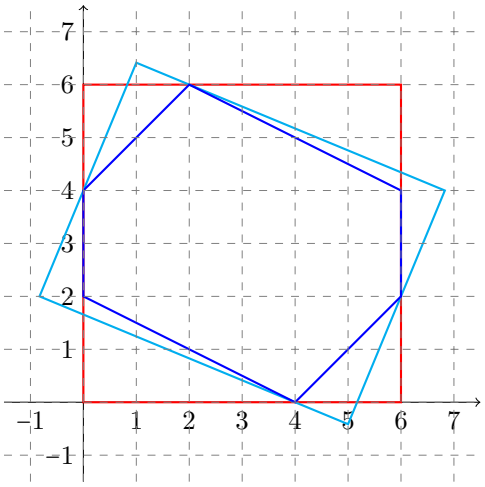


Figure 5.2: The plots of the zonotope conversion. The outline of the source zonotope is depicted in blue which is also the result of the exact conversions $Z \rightarrowtail V$, $Z \rightarrowtail H$ and $Z \rightarrowtail SF$, while the lines in cyan represent the over-approximation to an H-polytope/support function. The conversion to a box resulted in the red hyperrectangle.

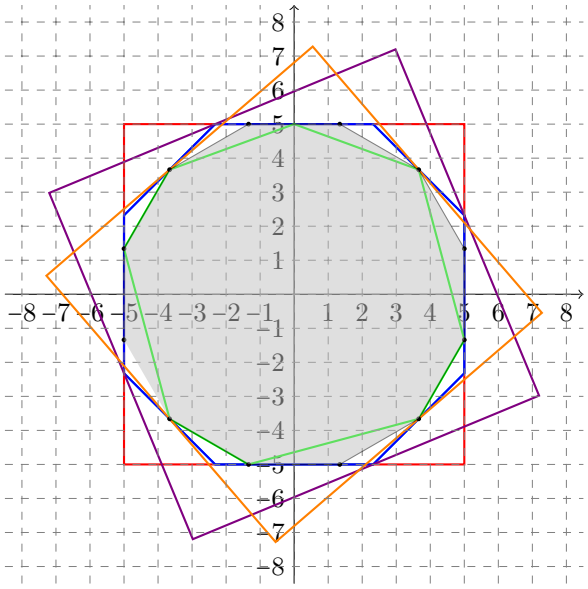


Figure 5.3: Output plots of the support function conversion using eight template directions. The black vertices and gray area represent the source object. Further displayed are the results of transformations $SF \twoheadrightarrow H$ and $SF \twoheadrightarrow V$ which both obtained the same result (blue object), in addition to the results of conversions $SF \twoheadrightarrow B$ (red object), $SF \dashrightarrow V$ (green object) and $SF \twoheadrightarrow Z$, where the violet ORH denotes the result of the normal conversion and the orange object depicts the output of the alternative conversion method.

Conversion Procedure	Avg. Runtime (in ms)		
	$u = 8$	$u = 12$	$u = 16$
$V \rightarrowtail H$	15.58	48.46	**160.04**
$V \twoheadrightarrow H$	0.42	2.28	**1.22**
$V \twoheadrightarrow B$	« 1	« 1	« 1
$V \twoheadrightarrow Z$	0.38	2.32	**2.82**
$V \rightarrowtail SF$	15.12	37.3	161.9
$H \rightarrowtail V$	17.8	135.14	**851.52**
$H \twoheadrightarrow B$	8.82	70.72	**547.68**
$H \twoheadrightarrow B$ (A)	0.92	2.14	**3.02**
$H \twoheadrightarrow Z$	19.36	168.22	**869.08**
$H \rightarrowtail SF$	« 1	« 1	« 1
$B \rightarrowtail V$	« 1	« 1	« 1
$B \rightarrowtail H$	« 1	« 1	« 1
$B \rightarrowtail Z$	« 1	« 1	« 1
$B \rightarrowtail SF$	« 1	« 1	« 1
$Z \rightarrowtail V$	« 1	« 1	« 1
$Z \rightarrowtail H$	0.96	1.08	0.9
$Z \twoheadrightarrow B$	« 1	« 1	« 1
$Z \rightarrowtail SF$	0.62	0.96	0.9
$SF \twoheadrightarrow H$	105.82	156.74	183.64
$SF \twoheadrightarrow V$	116.32	198.92	270.74
$SF \dashrightarrow V$	5.9	17.9	**34.32**
$SF \twoheadrightarrow B$	0.84	2.02	2.8
$SF \twoheadrightarrow Z$	145.06	185.9	**255.7**
$SF \twoheadrightarrow Z(A)$	13.66	20.4	**43.42**

Table 5.1: Results of the runtime analysis concerning all 24 conversion algorithms and the uniform template objects. The displayed values depict an average value regarding 50 computation runs (in milliseconds). The alternative version of an algorithm is denoted by (A). An entry « 1 is to be interpreted that the average time was much smaller than 1 ms.

5.1.2 General Analysis - Runtime

In order to give an indication about the general runtime that is to be expected when applying the procedures to 3-dimensional objects of different size, I constructed template objects of all representations, as described before, for each $u \in \{8, 12, 16\}$, resulting in template objects based on 24, 36 and 48 template directions. These template objects were then transformed using the implemented procedures; the corresponding results are presented via Table 5.1:

The contained values represent average values out of 50 computation runs and the conversions of support functions were conducted with the default eight template directions. Bold values are of most relevance for the following paragraphs, while red values cannot be explained by me, as certain procedures have less runtime for more complex objects.

As more complex objects are usually encountered during reachability analysis and

the simpler objects in the context of this experiment basically show the same results , I only consider the runtimes for template objects that were constructed with parameter $u = 16$ in the following subsections that analyse the obtained data.

Conversion of Polyhedra

As expected, the exact conversions between H-representation and V-representation are expensive, as they have exponential complexity, what is worth noting though is that an exact conversion $V \rightarrowtail H$ (160.04 ms avg.) is much cheaper for complex objects than vice versa (851.52 ms avg.), at least with the current implementation. The transformations $H \twoheadrightarrow B$ and $H \twoheadrightarrow Z$, which also utilise this exact conversion, therefore show long runtimes in comparison to the other algorithms as well (547.68 ms avg. and 869.08 ms avg. respectively).

Furthermore, when dealing with a V-polytope as the source object, the over-approximation of an H-polytope with an ORH is obviously done a lot faster than conducting an exact conversion; the results here show the in fact huge differences very well: With only 1.22 ms avg. needed for a computation of a fitting ORH in comparison to the already mentioned 160 ms avg. for the exact approach, the over-approximation was more than a hundred times faster. It is also worth noting that the additional computations that are required for the calculation of a zonotope out of the ORH do not seem to have a big impact on the computation time, as the over-approximative conversion from V-polytope to zonotope required only 2.82 ms in average.

On top of that, regarding the two developed procedures for obtaining an over-approximative hyperrectangle of an H-polytope, the alternative algorithm clearly outclasses the naive approach: Both algorithms provide exactly the same precision, as they both always calculate the intervall hull, but the alternative approach is by far more efficient, as it needed merely 3.02 ms avg. in contrast to the 547.68 ms avg. provided by the naive procedure. This result is no surprise, as the naive approach has exponential complexity, while the alternative solves a small number of optimisation problems that only scale with the dimension. It is therefore adviced to exclusively use the alternative algorithm.

Conversion of Hyperrectangles

With the exception of the naive approach for the calculation of the interval hull of an H-polytope, all conversions from or to a hyperrectangle were conducted very fast, in fact, most of the approaches took a time of much less than 1 ms in average. This is also unsurprising when considering the very simple geometry of boxes.

Conversion of Zonotopes

The conversion of the utilised template zonotopes was also done very efficiently. As all template zonotopes are ORHs, the results show that, when converting with an ORH as the source object, conversion to other representations is marginally slower than with simple boxes, at least in the third dimension.

Conversion of Support Functions

Concerning the conversion of support functions, the result values show that the implementations that compute boundary points were faster than the algorithms that mainly compute supporting hyperplanes: The under-approximation from support function to V-polytope and the alternative approach for a conversion to a zonotope both featured an average computation time of less than 50 ms which is a lot faster than the rest of the procedures (aside from the conversion to a box).

In contrast to the alternative approach, the default version for a zonotope construction in this context is with 255.7 ms avg. about five times as slow. This can be explained with the default conversion algorithm being composed of many costly smaller conversions. As the output objects concerning both algorithms in general differ, it is required to take a look at the precision that these algorithms provide before judging solely based on the runtime. A comparison precision-wise is presented with Subsection 5.2.2.

5.2 Specific Experiments and Plots

While the previous section provides a look at all implemented procedures, I subsequently present two smaller experiments that have the purpose of examining additional properties of only a few of the introduced techniques:

Subsection 5.2.1 investigates the efficiency of oriented rectangular hulls, while Subsection 5.2.2 compares both of the introduced approaches for transforming support functions into zonotopes.

5.2.1 Precision of Oriented Rectangular Hulls

In Section 5.1 we saw that most conversions to and from oriented rectangular hulls can be efficiently computed, but it is not obvious how well ORHs enclose more irregular objects. To make things clearer, I prepared four V-polytopes that were overapproximated by ORHs (cf. Subsection 4.2.1). The resulting output plots can be seen in Figure 5.4, along with the vertices of the source objects and their interval hulls. I discuss the obtained results in the following:

A scenario, in which the obtained ORH is exactly the interval hull and therefore brings no advantage with it whatsoever, is shown in Figure 5.4a. On top of that, the vertices in this situation have a strictly symmetrical distribution concerning the arithmetic mean of the vertices, which means that there are multiple valid matrices U (in this case only two) that could be obtained by the singular value decomposition. The second possible ORH (green dashed line) encloses the set of vertices much better, but was not computed by the algorithm. It is worth noting though that such symmetric objects are rarely encountered in praxis.

Regarding the rest of the plots, the ORH provides better precison than the interval hull, with substantially better approximations in Figure 5.4c and Figure 5.4d.

Figure 5.4d is particularly interesting, as it shows that ORHs can also be used for the fast construction of valid full-dimensional H-polytopes (and zonotopes) when only a few points of a line are given. This situation can occur when the intersection set of e.g. a flow pipe segment with a guard is converted to a V-polytope: The resulting polytope in this case possibly only consists of two points; for higher dimensions, it

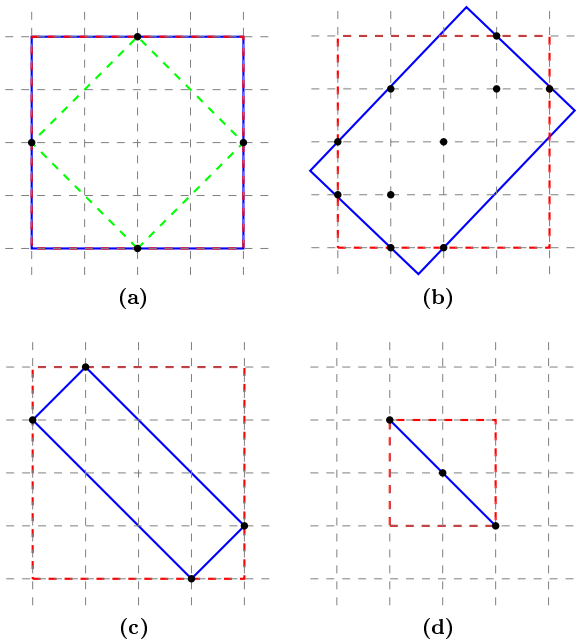


Figure 5.4: Plots for four ORH-computations. The black points represent the vertices of the source V-polytope, while the blue objects depict the resulting ORHs; the dashed lines in red visualise the interval hull of the source vertices, whereas the dashed geometry in green indicates the optimal ORH regarding the scenario depicted in (a).

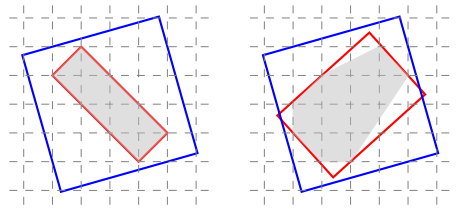
is actually not obvious how to construct a full-dimensional H-polytope out of two points. We now know that ORHs provide an efficient way of doing so.

5.2.2 $SF \twoheadrightarrow Z$ - Precision

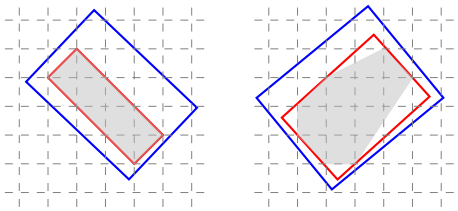
With Subsection 5.1.2 indicating that the alternative approach for the conversion of support functions to zonotopes that makes use of boundary points is a lot faster than its counterpart, this section provides a comparison of both algorithms in terms of output precision:

Two of the four V-polytopes, that were over-approximated with ORHs in the previous experiment, were converted to support functions in the context of this experiment beforehand. Each object was then converted according to both algorithms for 4,5,6,7 and 8 template directions, yielding a total of 10 output plots that are presented with Figure 5.5.

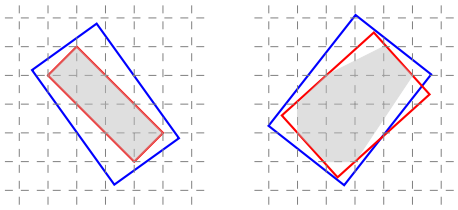
All findings here show clearly that the alternative approach using boundary points is superior to its twin in terms of precision as well: The boundary point algorithm obtained a tight over-approximation for both objects with only 4 template directions already, in fact, it computed exactly the source object in case of Ω_1. An increase of the number of directions had no further effect in case of Ω_1, while the overall precision basically stayed about the same with ORHs of slightly different alignment obtained with increasing number of directions. What is curious though is that the precision of the approximations actually suffered a bit from the increase of template directions be-



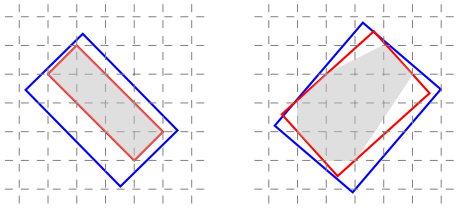
(a) Ω_1, 4 directions **(b)** Ω_2, 4 directions



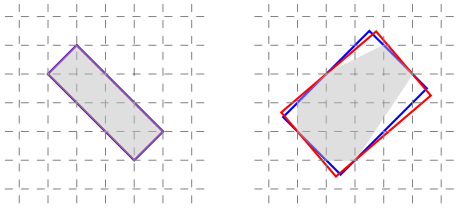
(c) Ω_1, 5 directions **(d)** Ω_2, 5 directions



(e) Ω_1, 6 directions **(f)** Ω_2, 6 directions



(g) Ω_1, 7 directions **(h)** Ω_2, 7 directions



(i) Ω_1, 8 directions **(j)** Ω_2, 8 directions

Figure 5.5: Comparison of the two different conversion algorithms from support function to zonotope. The convex sets Ω_1 and Ω_2 that are represented by support functions are depicted by gray areas and were converted to zonotopes using both approaches. The red objects visualise the results of the alternative approach that uses boundary points, while the blue objects represent the output of the default technique that builds an H-polytope with supporting hyperplanes instead.

yond 5. This is probably due to the computation of unfavourable template directions, beginning with 6 template directions (cf. Figure 5.5f).

The default procedure on the other hand often resulted in over-approximations of such sizes (cf. e.g. Figure 5.5a) that even a conversion to a hyperrectangle would have been more precise (and much cheaper). With growth of the number of template directions, the results obtained by the default algorithm slowly approximated the at all times better or equally good output objects of the alternative approach.

It is therefore recommended to use the boundary point computation for the calculation of an over-approximating zonotope. But as this is not always possible due to issues already discussed, the default algorithm is possibly still useful in scenarios in which the boundary point computation works incorrectly, at least when being conducted with enough template directions. What is "enough" though for the underlying geometry is hardly possible to tell without additional information.

Chapter 6

Conclusion

This final chapter concludes this bachelor thesis by summarising and discussing the results, and emphasising possible future work.

6.1 Summary

In the course of this thesis, I developed and implemented twenty-four conversion algorithms concerning five popular geometrical and symbolical state set representations against the backdrop of reachability analysis for hybrid systems. With especially flow pipe computation in mind, these procedures were designed to ensure that each d-dimensional hyperrectangle, zonotope, support function and convex polytope in either H-representation or V-representation could be transformed into any of the other representations efficiently, giving the opportunity to change representations at will during computation and thus to fully exploit the individual advantages of the various representations.

The presented techniques support all Euclidian dimensions and make use of a variety of concepts, with the most notable being the construction of over-approximating oriented rectangular hulls and the currently still restrictedly working boundary point computation of support functions.

My conducted evaluation of the implemented procedures shows a large disparity in terms of precision and necessary computation time among many of the algorithms, confirming the trade-off between computation time and output quality that is usually present in the research field of computer science. Oriented rectangular hulls were recognised for providing a good balance in these terms. Concerning the two conversion scenarios for which I implemented two different approaches, the affected procedures were compared and the superior algorithm was identified in both cases.

6.2 Discussion

The general objective of providing efficient conversion procedures could be achieved for the most part: Nearly all of the introduced algorithms work correctly and obtain fairly good results in a reasonable time window, with respect to the individual difficulty of the featured conversions. However, some of the conversions (like

the over-approximative conversion from H-polytope to zonotope) have proven to be problematic to realise in an efficient manner.

Furthermore, altough there was a lot of effort put into the under-approximation of support functions with V-polytopes, I was not able to find a computation method that works unrestrictedly. Nevertheless, the boundary point computation of support functions is by far not trivial and also currently not covered to its full extent in literature.

It is also unfortunate that I was not able to present an actual exemplary application of my algorithms in the bigger context of hybrid systems reachability analysis due to the only limited time that I had at my disposal.

So all in all, I claim that my developed procedures provide an overall reasonable efficiency and make a valuable addition to the HyPro project, while there is still room for improvement, as summarised by the following final section.

6.3 Future work

Multiple possible topics for future research were already indicated in the course of this thesis and are closer specified in the following:

1. First and foremost, solving the current issues with the boundary point computation for support functions would remove the present restrictions for some of my algorithms. Achieving this goal could involve the improving of the intersection operation regarding support functions and/or deriving fitting algebraical representations out of support functions in order to be able to compute their directional derivatives like described in [GK98].

2. The already mentioned zonotope construction problem (cf. [Fuk04]) may also be worth examining, as the ability to efficiently compute an H-polytope out of a general zonotope (and vice versa) would yield improvements for the corresponding algorithms.

3. Subsection 5.2.1 shows that the current ORH-computation is currently not able to choose the most beneficial orientation in case there are mutliple valid singular value decompositions. With the consideration of additional properties, it should be possible to choose the best orientation.

4. Last but not least, nearly all of my conversions of support functions make use of template directions. While this is fitting in scenarios in which there is no information about the underlying geometry, it could be worthwile to explore the possibilities of choosing sensible directions, if there is more information at hand. On top of that, enabling the user to specify his own evaluation directions is another possible way of enhancing the existing procedures.

Bibliography

[ABD+00] Eugene Asarin, Olivier Bournez, Thao Dang, Oded Maler, and Amir Pnueli. Effective Synthesis of Switching Controllers for Linear Systems. In *Proceedings of the IEEE*, pages 1011–1025. IEEE, 2000.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138:3–34, 1995.

[AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183 – 235, 1994.

[AD14] M. Althoff and J. M. Dolan. Online Verification of Automated Road Vehicles Using Reachability Analysis. *IEEE Transactions on Robotics*, 30(4):903–918, 2014.

[ADI02] Rajeev Alur, Thao Dang, and Franjo Ivančić. *Hybrid Systems: Computation and Control: 5th International Workshop*, chapter Reachability Analysis of Hybrid Systems via Predicate Abstraction, pages 35–48. Springer, 2002.

[AMHS01] E. Ábrahám-Mumm, U. Hannemann, and M. Steffen. Verification of Hybrid Systems: Formalization and Proof Rules in PVS. In *Proceedings of IEEE Engineering of Complex Computer Systems*, pages 48–57. IEEE Computer Science Press, 2001.

[ASB08] Matthias Althoff, Olaf Stursberg, and Martin Buss. Verification of Uncertain Embedded Systems by Computing Reachable Sets Based on Zonotopes. In *Proceedings of the 17th IFAC World Congress*, 2008.

[ASB09] Matthias Althoff, Olaf Stursberg, and Martin Buss. Computing Reachable Sets of Hybrid Systems Using a Combination of Zonotopes and Polytopes. *Nonlinear Analysis: Hybrid Systems*, 3, 2009.

[BMP99] Olivier Bournez, Oded Maler, and Amir Pnueli. Orthogonal Polyhedra: Representation and Computation. *Lecture Notes in Computer Science*, 1569:46–60, 1999.

[CÁS13] X. Chen, E. Ábrahám, and S. Sankaranarayanan. Flow*: An Analyzer for Non-linear Hybrid Systems. In *Conference on Computer Aided Verification*, volume 8044, pages 258–263, 2013.

[CBGV12] Pieter Collins, Davide Bresolin, Luca Geretti, and Tiziano Villa. Computing the Evolution of Hybrid Systems Using Rigorous Function Calculus. In *Analysis and Design of Hybrid Systems*, pages 284–290. IFAC-PapersOnLine, 2012.

[CDL09] Edmund Clarke, Alexandre Donzé, and Axel Legay. *Hardware and Software: Verification and Testing: 4th International Haifa Verification Conference*, chapter Statistical Model Checking of Mixed-Analog Circuits with an Application to a Third Order Δ–Σ Modulator, pages 149–163. Springer, 2009.

[CFH+03] Edmund M. Clarke, Ansgar Fehnker, Zhi Han, Bruce H. Krogh, Joël Ouaknine, Olaf Stursberg, and Michael Theobald. Abstraction and Counterexample-Guided Refinement in Model Checking of Hybrid Systems. *International Journal of Foundations of Computer Science*, 14(4):583–604, 2003.

[CK98] A. Chutinan and B. H. Krogh. Computing Polyhedral Approximations to Flow Pipes for Dynamic Systems. In *Proceedings of the 37th Annual Conference on Decision and Control*, volume 2, pages 2089–2094. IEEE, 1998.

[DB] Mireille Broucke Department and Mireille Broucke. Reachability Analysis of Hybrid Systems with Linear Dynamics. `http://www3.nd.edu/~mtns/papers/13040_2.pdf`.

[DM07] Alexandre Donzé and Oded Maler. Systematic Simulation Using Sensitivity Analysis. In *Hybrid Systems: Computation and Control*, volume 4416, pages 174–189, 2007.

[Dun89] G.H. Dunteman. *Principal Components Analysis*. Number 69. SAGE Publications, 1989.

[Edd77] William F. Eddy. A new convex hull algorithm for planar sets. *ACM Transactions on Mathematical Software*, 3(4):398–403, 1977.

[Egg14] Andreas Eggers. *Direct Handling of Ordinary Differential Equations in Constraint-Solving-Based Analysis of Hybrid Systems*. PhD thesis, Universität Oldenburg, Germany, 2014.

[FGD+11] Goran Frehse, Colas Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. *Computer Aided Verification: 23rd International Conference*, chapter SpaceEx: Scalable Verification of Hybrid Systems, pages 379–395. Springer, 2011.

[Fuk] Komei Fukuda. Polytope Examples (Fukuda) – Zonotopes. `http://www.cs.mcgill.ca/~fukuda/760B/handouts/expoly3.pdf`.

[Fuk04] Komei Fukuda. From the Zonotope Construction to the Minkowski Addition of Convex Polytopes. *Journal of Symbolic Computation*, 38:1261–1272, 2004.

[Gal08] Jean Gallier. Notes on Convex Sets, Polytopes, Polyhedra, Combinatorial Topology, Voronoi Diagrams and Delaunay Triangulations. *arXiv preprint arXiv:0805.0292*, 2008.

[GG09] Colas Guernic and Antoine Girard. *Computer Aided Verification: 21st International Conference*, chapter Reachability Analysis of Hybrid Systems Using Support Functions, pages 540–554. Springer, 2009.

[Gir05] Antoine Girard. *Hybrid Systems: Computation and Control: 8th International Workshop*, chapter Reachability of Uncertain Linear Systems Using Zonotopes, pages 291–305. Springer, Berlin, Heidelberg, 2005.

[GK98] Pijush K. Ghosh and K. Vinod Kumar. Support Function Representation of Convex Bodies, Its Application in Geometric Computing, and Some Related Representations. *Computer Vision and Image Understanding*, 72(3):379–403, 1998.

[GLG08] Antoine Girard and Colas Le Guernic. *Hybrid Systems: Computation and Control: 11th International Workshop*, chapter Zonotope/Hyperplane Intersection for Hybrid Systems Reachability Analysis, pages 215–228. Springer, 2008.

[GNZ03] Leonidas J. Guibas, An Nguyen, and Li Zhang. Zonotopes As Bounding Volumes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 803–812. Society for Industrial and Applied Mathematics, 2003.

[HKPV95] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's Decidable About Hybrid Automata? In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, pages 373–382. ACM, 1995.

[HR98] Thomas A. Henzinger and Vlad Rusu. *Hybrid Systems: Computation and Control: First International Workshop*, chapter Reachability Verification for Hybrid Automata, pages 190–204. Springer, 1998.

[HyC] HyCreate: A Tool for Overapproximating Reachability of Hybrid Automata. `http://stanleybak.com/projects/hycreate/hycreate.html`.

[KGCC15] S. Kong, S. Gao, W. Chen, and E. M. Clarke. dReach: δ-Reachability Analysis for Hybrid Systems. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035, pages 200–205, 2015.

[KV00] Alexander B. Kurzhanski and Pravin Varaiya. Ellipsoidal Techniques for Reachability Analysis. In *Proceedings of the Third International Workshop on Hybrid Systems: Computation and Control*, pages 202–214. Springer-Verlag, 2000.

[LG09] Colas Le Guernic. *Reachability Analysis of Hybrid Systems with Linear Continuous Dynamics*. PhD thesis, Université Grenoble I – Joseph Fourier, 2009.

[LGG10] Colas Le Guernic and Antoine Girard. Reachability Analysis of Linear Systems Using Support Functions. *Nonlinear Analysis: Hybrid Systems*, 4, 2010.

[MBT01] Ian Mitchell, Alexandre M. Bayen, and Claire J. Tomlin. *Hybrid Systems: Computation and Control: 4th International Workshop*, chapter Validating a Hamilton-Jacobi Approximation to Hybrid System Reachable Sets, pages 418–432. Springer, 2001.

[MKC09] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 2009.

[Neu] Arnold Neumaier. Taylor Forms—Use and Limits. *Reliable Computing*, 9(1):43–79.

[PQ08] André Platzer and Jan-David Quesel. KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description). In *International Joint Conference on Automated Reasoning*, volume 5195, pages 171–178, 2008.

[RS07] Stefan Ratschan and Zhikun She. Safety Verification of Hybrid Systems by Constraint Propagation-based Abstraction Refinement. *ACM Trans. Embed. Comput. Syst.*, 6(1), 2007.

[SK03] Olaf Stursberg and Bruce H. Krogh. Efficient Representation and Computation of Reachable Sets for Hybrid Systems. In *Proceedings of the 6th International Conference on Hybrid Systems: Computation and Control*, pages 482–497. Springer, 2003.

[Var00] Pravin Varaiya. *Verification of Digital and Hybrid Systems*, chapter Reach Set Computation Using Optimal Control, pages 323–331. Springer, 2000.

[WT97] H. Wong-Toi. The Synthesis of Controllers for Linear Hybrid Automata. In *Proceedings of the 36th IEEE Conference on Decision and Control*, volume 5, pages 4607–4612. IEEE, 1997.

[Zie95] G. M. Ziegler. *Lectures on Polytopes*, volume 152 of *Graduate Texts in Mathematics*. Springer, 1995.