Calcul d'Atteignabilité des systèmes hybrides avec des fonctions de support

Rajarshi Ray

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Mathématiques et Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Mr. Rajarshi Ray

Thèse dirigée par **Dr. Oded Maler**
et codirigée par **Dr. Goran Frehse**

préparée au sein **Verimag**
et de **Ecole Doctorale Mathématiques, Sciences Et Technologies De L'information, Informatique**

Reachability Analysis of Hybrid Systems Using Support Functions

Thèse soutenue publiquement le **29th May, 2012**,
devant le jury composé de :

Prof. Eugene Asarin
Université Paris Diderot - Paris 7, Rapporteur
Prof. Radu Grosu
Vienna University of Technology, Rapporteur
Prof. Andreas Podelski
University of Freiburg, Examinateur
Dr. Colas Le Guernic
DGA, France, Examinateur
D.R. Oded MALER
CNRS, Directeur de thèse
MCF Goran FREHSE
Université Joseph Fourier Grenoble 1, Co-Directeur de thèse

Contents

List of Figures

List of Tables

Acknowledgements

My gratitude to my thesis advisor, Goran Frehse for his tremendous support and encouragement during the course of my thesis. Goran has been my first point of contact for almost all the help I needed during my stay in Verimag, be it academic or non-academic. I am grateful to my thesis director Oded Maler for all this help with my thesis. My thanks also to the TEMPO team members.

My gratitude to Prof. Eugene Asarin and Prof. Radu Grosu for helpful comments on my thesis manuscript, Prof. Andreas Podelski and Dr. Colas Le Guernic for examining my work.

It has been a pleasure working with the SpaceEx team members. I would like to particularly thank Scott Cotton, Olivier Lebeltel and Manish Goyal for helpful discussions and exchange of thoughts.

My thanks to all my friends and colleagues at Verimag with whom I have spent a wonderful time. My officemates at Bureau 41, Selma Saidi, Julien Le Guernic and Marion Daubignard who inspired each other during the course of Ph.D.

My thanks to Tayeb Bouhadiba for his help on latex and regarding the technical arrangements for my thesis defense at the CTL auditorium at Verimag.

I am grateful to my friends, Pranav Tendulkar, Vrushali Tendulkar, Sanjay Rawat, Dipti Dahiya, Ananda Basu, Priyadarshini Basu and Parantapa Goswami who arranged a wonderful post defense treat and prepared delicious Indian recipes.

I am grateful to my family for the support and inspiration during the course of my Ph.D. thesis.

Chapter 1

Introduction

1.1 The Need for Formal Methods

As a matter of fact, human beings are becoming more and more dependent on technology products, large and small, software and hardware. Mobile phones, electronic devices like tablet computers and laptops, software services like the email and social networking sites, software applications like google calender, operating systems, Internet technology and more have become our daily needs. We can sense that this human dependence on technology is going to increase in the future. Greater dependence on technology compels us to establish their correctness or perfectness. Imperfections can be tolerated for not so critical applications but not otherwise. We can tolerate if the operating system in our laptop crash when an audio player is run or an email sent to one person ends up in the inbox of someone else, but we cannot tolerate slightest error in, for example, the *Traffic Alert and Collision Avoidance System* for air traffic control which might lead to a mid air collision. In fact, we are becoming less and less tolerant regarding technology errors.

There had been incidences of technology failures in the past in the realm of critical systems. The explosion of Ariane 5 rocket in June 4, 1996, 40 seconds after its take off is a recent example of technology failure whose cause, according to an inquiry report [Lio96], is due to a software design error in the onboard computer system. Even more recently, the Space Shuttle Columbia disaster on 1 February, 2003 which resulted in the death of all the seven NASA astronauts on board is yet another example of intolerable technology failure. The Columbia Accident Investigation Board (CAIB) reported the loss of Columbia as a result of damage sustained during launch when a piece of foam insulation the size of a small briefcase broke off from the Space Shuttle external tank under the aerodynamic forces of launch. The debris struck the leading edge of the left wing, damaging the Shuttle's thermal protection system (TPS), which shields it from the intense heat generated from atmospheric compression during re-entry. NASA's original shuttle design specifications stated that the external tank was not to shed foam or other debris. NASA launches in the past reported debris strikes but they were not taken as a security threat and the design flaw was accepted as inevitable and unresolvable. This deviation from the original design specification is blamed for the disaster [Dis]. These incidences shed light on the importance of *design specification* and *design validation*.

Formal methods are to counter technology imperfections to as much extent as possible or to remove imperfections which really matter. Formal methods are mathematical

techniques for the specification, development and verification of software and hardware systems. The goal of formal methods is to contribute to the reliability and robustness of a software or hardware system. The application of formal methods in real world is seemingly increasing and with the advent of more powerful tools which are scalable, the future looks brighter for formal methods based techniques in design validation.

1.2 Model Checking

The foreword by Amir Pneuli to the introductory book on model checking [JGP99] is a beautiful insight to the problem of *Design Validation - Ensuring the correctness of the design at the earliest possible stage*. Quoting from the foreword - "*The major obstacle to "help computers help us more" and to relegate to these helpful partners even more complex and sensitive tasks is not inadequate speed and unsatisfactory raw computing power in the existing machines, but our limited ability to design and implement complex systems with sufficiently high degree of confidence in their correctness under all circumstances*".

Simulation, testing and deductive verification are traditional approaches to gain greater confidence on systems. While simulation is carried out on the model of a design, testing is performed on the actual design itself. Deductive reasoning is a mathematical proof system where correctness of systems are proved with axioms and proof rules. We know that simulation and testing are inadequate in establishing total confidence of the design under validation because they are not exhaustive checks. Deductive verification has advantages and disadvantages of its own. Deductive verification can be extremely expensive at times.

Model checking is an automatic technique for verifying finite state systems. The procedure normally uses an exhaustive search of the state space of the system to determine if a specification is true or not. There are broadly two types of system properties those are checked with model checking algorithms, namely *safety* properties and *liveness* properties. Safety properties are properties which specify that nothing bad occurs. Liveness properties are properties which specify that something good eventually occurs. There are model checking algorithms which are reasonably efficient and allows for its automation. Infinite state systems can also be model checked with abstractions which constructs finite symbolic states from the infinite state space. Model checking, however, suffers from the *state space explosion* problem which arises due to the exponential increase in the number of explicit states of a system. State space explosion problem can be tackled to some extent with model abstractions, use of efficient data structures, heuristics and symbolic representation of states.

Applying model checking for design validation mainly involves three steps: (1) Modeling, (2) Specification and (3) Verification. Modeling is the process of formalizing a design with a mathematical model. A model is sometimes abstracted to hide unnecessary details and to make it within analysis limits and trying to cope with the state space explosion problem. Specification means formally stating the properties that the design must satisfy. *Temporal Logic* is used to specify properties over time for example. Verification is the process of automatically checking if the given specifications are satisfied by the design under validation. If a specification is found to be violated, a counter example is expected to be returned by the model checking algorithm showing the design behavior that violated the given specification. When working with abstraction of models, it becomes necessary

to check if the generated counter example is spurious.

Examples of some model checkers are SPIN [Hol97], UPPAAL [BLL+96], Kronos [BDM+98], HyTech [HHWT97] and PHAVer [Fre08]. SPIN is a model checker for distributed software systems against LTL specifications. UPPAAL is a model checker for real time systems modeled with timed automata. Kronos is similarly a model checker for real time systems modeled with timed automata against TCTL specifications. HyTech and PHAVer are model checkers for hybrid systems modeled as linear hybrid automata.

1.3 Hybrid Automata

Hybrid automata are a modeling formalism that combines discrete events with continuous variables that change over time [ACH+95], [Hen96]. Formally, a hybrid automaton $\mathcal{H} = (Loc, Var, Lab, Trans, Flow, Inv, Init)$

consists of the following elements:

- a graph whose vertices, called *locations*, are given by a finite set Loc, and whose edges, called *discrete transitions*, are given by a finite set $Trans$;
- a finite set of real-valued variables Var. A *state* of the automaton consists of a location and a value for each variable (formally described as a *valuation* over Var). The set of all states of the automaton is called its *state space*. To simplify the presentation, we assume that the state space is $Loc \times \mathbb{R}^n$, where n is the number of variables. We will also simply write x to denote the name of the variable x or its value according to the context;
- for each location, the variables can only take values in a given set called *invariant*. The invariants are given by $Inv \subseteq Loc \times \mathbb{R}^n$;
- for each location, the change of the variables over time is defined by its time-derivative that must be in a given set $Flow \subseteq Loc \times \mathbb{R}^n \times \mathbb{R}^n$. For example, if the system is in a location l, a variable x can take the values of a function $\xi(t)$ if at each time instant t, $(l, \dot{\xi}(t), \xi(t)) \in Flow$, where $\dot{\xi}(t)$ denotes the derivative of $\xi(t)$ with respect to time;
- the *discrete transitions* $Trans \subseteq Loc \times Lab \times 2^{\mathbb{R}\times\mathbb{R}} \times Loc$ specify instantaneous changes of the state of the automaton. A transition (l, α, μ, l') signifies the system can instantaneously jump from any state (l, x) to any state (l', x') if $x' \in Inv(l')$ and $(x, x') \in \mu$. Every transition has a *synchronisation* $\alpha \in Lab$ that is used to model the interaction between several composed automata. Intuitively, if two automata share a common α, transitions with this can only be executed in unison, i.e., by simultaneous execution of a transition with this label in both automata. The relation μ is called the *jump relation* of the transition;
- A set of states $Init \subseteq Loc \times \mathbb{R}^n$ specifies the *initial states* from which all behavior of the automaton begins.

Figure 1.1 shows a bouncing ball modeled as a hybrid automaton. The ball's velocity change with time constitutes the continuous aspect and the discrete change in the velocity

at strikes with the ground constitutes the discrete aspect. The only location *always* in this example has a location invariant $x \geq 0$ and a flow equation $\dot{x} = v \ \& \ \dot{v} = -g$. The variable x stands for the position of the ball and v stands for its velocity. There is a self transition with the label *hop*. The transition has a guard given by $x \leq 0 \ \& \ v < 0$ and an assignment $v' = -c.v$. x, v are the continuous variables of this system and c, g are constants. v' in the transition assignment denotes the new value of the velocity after the transition has taken place. This hybrid automaton defines a two dimensional hybrid system because it models the behavior of two continuous variables.

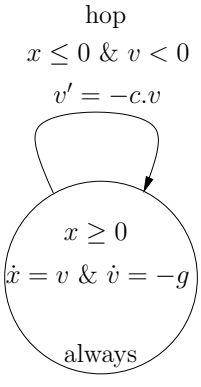


Figure 1.1: Bouncing ball modeled with a hybrid automaton.

In the next section, we introduce the notion of reachability.

1.4 Reachability

A reachable state of a hybrid automaton is a valuation to the continuous variables which is possible under the dynamics of the system. The dynamics of the system defines the evolution of the continuous variables with time. When we say a reachable state of a hybrid automaton, it is meaningful only when it is defined relative to an initial state in *Init*. The execution of a hybrid automaton results in continuous change (flows) and discrete change (jumps). A result of executing a hybrid automaton from an initial state x_0 is a trajectory, say π_{x_0}.

A trajectory is the path constituting all the states starting from the initial state that the system can take under its dynamics. A trajectory is unique for a given initial state if the system under consideration is deterministic, having empty input set $\mathcal{U}$. Given a time instant t and an initial state x_0, $\pi_{x_0}(t)$ denotes the state of the trajectory initiated from x_0 at time t.

All reachable states of a hybrid automaton constitute its reachable set. Computing the reachable set is what we call as reachability computation. The reachable set can also be seen as the union of all trajectories of the hybrid automata.

$$\mathcal{R} = \{x \in \mathbb{R}^n \mid \exists x_0 \in \mathit{Init},\ t \in \mathbb{R} \ \mathit{such\ that}\ \pi_{x_0}(t) = x\}$$

If there is a finite number of initial states and the hybrid automaton is completely deterministic then computing all the trajectories and taking their union would give us the

reachable set. Unfortunately, there is often infinite number of initial states in $Init \subseteq \mathbb{R}^n$ and hence one would ideally need infinite number of trajectory computations to get the reachable set which is infeasible.

A trajectory of a dynamical system can be computed with numerical simulations for a given start state and input using numerical integration. Simulation is handy for design validation up-to a certain degree of confidence. There are some fast simulators available for dynamical systems like the MATLAB Simulink [Sim] which has become an industrial de facto standard for model based development of complex systems. Simulations can also provide the designers with an overall idea of the reachable set by choosing some clever simulation start points like the corner cases. However, simulations in general cannot guarantee safety or liveness properties. Reachable set on the other hand if computed can guarantee safety and liveness properties and that is the main motivation. For a continuous initial state $Init$ and input set $\mathcal{U}$, one need to perform infinite number of simulations in theory to check all possible behaviors. Hence, we can deduce that reachability computation with numerical simulations is not a feasible solution.

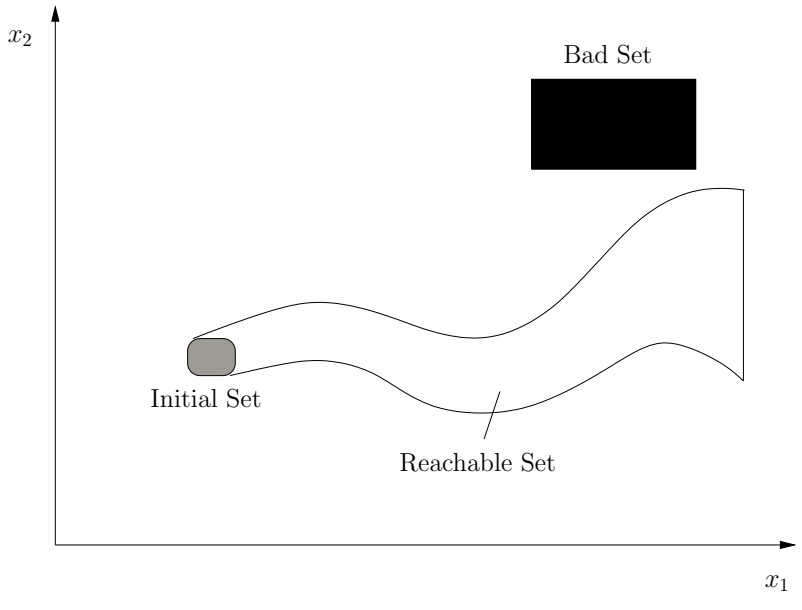


Figure 1.2: Illustrating safety verification with set based reachability analysis. The Gray set denotes the initial states of the system and the Black set denotes the bad or the error states. Reachable set denotes all possible states taken by the system. Empty intersection of the reachable set with the bad set implies safety.

A hybrid automaton consists of potentially infinitely many states. For an algorithmic analysis, we need a finite representation of the infinite state space and that is done through symbolic state representation. We define a symbolic state to be a pair of a discrete set and a continuous set. Semantically, the discrete set is a set of locations and the continuous set gives the possible valuations of the continuous variables of the hybrid system in the location(s). For example, if we have locations d_1, d_2 and d_3 in a discrete set and a continuous set is given by a unit hypercube in $\mathbb{R}^n$, the symbolic state comprising of the pair of this discrete and continuous set represents all hybrid automaton states $(v_1, ..., v_n) \in$ unit hypercube when in location d_1 or d_2 or d_3. Reachability computation will consist of searching exhaustively for all symbolic states till the fixpoint is reached. For hybrid automata where fixpoint do not exist, we could compute bounded reachability

which is to compute all reachable states up-to a time bound T.

$$\mathcal{R}(T) = \{x \in \mathbb{R}^n \mid \exists x_0 \in Init,\ t \in [0,T]\ \textit{such that}\ \pi_{x_0}(t) = x\}$$

Similarly, we also might be interested in the states of the system between a time interval which is defined as follows:

$$\mathcal{R}(t_1,t_2) = \{x \in \mathbb{R}^n \mid \exists x_0 \in Init,\ t \in [t_1,t_2]\ \textit{such that}\ \pi_{x_0}(t) = x\}$$

Theoretically, the reachability problem of a hybrid automaton $\mathcal{H}$ concerns with the question that - Is there a trajectory of $\mathcal{H}$ which starts in X_0 and ends in X_F? It is shown that for hybrid systems in general, the reachability problem is undecidable [ACH+95], [HKPV95]. The reachability problem for even most of the simpler classes of hybrid systems is shown to be undecidable. Undecidability has not kept the research community away from the subject of formal verification of hybrid systems though. Algorithms and heuristics have been developed and are still being developed to compute an over-approximation of the reachable set and then it is checked if this over-approximated set is safe, i.e., it does not intersect with the bad set. An over-approximated set $\mathcal{R}_{over}$ contains more states than the model actually reaches, i.e., $\mathcal{R} \subseteq \mathcal{R}_{over}$. Safety of the over-approximated set implies the safety of the design under validation. If there is an intersection of the over-approximated set with the bad set, unsafety is not implied however. The intersection with the bad set could be due to the over-approximation. Counter examples can be generated from the parts of the reachable set that intersect with the bad set. To identify if the obtained counter example is spurious, one can use the CEGAR approach (Counter Example Guided Abstraction Refinement) [CGJ+00] which refines the abstraction, i.e., improves the approximation to eliminate behaviors guided by the counter examples. Unsafety can be guaranteed by computing an under-approximated reachable set $\mathcal{R}_{under}$. An under-approximated reachable set contains less states than the actual reach set of the model, i.e., $\mathcal{R}_{under} \subseteq \mathcal{R}$. If the under-approximated reachable set intersect with the bad set, then the design under validation is guaranteed to be unsafe.

1.5 Thesis Scope and Outline

This thesis is broadly an attempt to attack the problem of *design validation* and to widen the existing horizon of the state of the art. Reachability analysis of hybrid systems is the focus of this thesis. We explore the use of *support functions* for the reachability analysis.

There are a number of classes of hybrid automata namely rectangular hybrid automata and linear hybrid automata (LHA). Each of this models a class of hybrid systems. In a LHA, for each variable the rate of change is constant. although this constant can be different in each location. The terms involved in the invariant, guard and assignments are required to be linear. Timed automaton [AD94] is a special case of LHA where the variables are clocks with rate of change always as 1.

The context of this thesis remains restricted to linear hybrid automata (LHA) and hybrid automata having affine continuous dynamics with uncertain inputs and affine maps on the discrete jumps. The type of location dynamics we are concerned with is as follows:

$$\dot{x}(t) = Ax(t) + u(t), \qquad u(t) \in \mathcal{U}, \tag{1.1}$$

where $x(t) \in \mathbb{R}^n$, A is a real-valued $n \times n$ matrix and $\mathcal{U} \subseteq \mathbb{R}^n$ is a closed and bounded convex set.

Transition assignments are of the form

$$x' = Rx + w, \qquad w \in \mathcal{W}, \tag{1.2}$$

where $x' \in \mathbb{R}^m$ the values after the transition, $R \in \mathbb{R}^m \times \mathbb{R}^n$ is the assignment map, and $\mathcal{W} \subseteq \mathbb{R}^n$ is a closed and bounded convex set of non-deterministic inputs.

These two classes of hybrid automata can model a wide range of systems in real life and can be used to approximate non-linear systems.

In chapter 2, we start with the presentation of a basic reachability algorithm for hybrid systems. In section 2.1, we present the concept of symbolic states and how they are used to efficiently compute the reachable set of infinite state systems. Section 2.2.1 presents some basic definitions, theorems and propositions on convex analysis that we frequently refer later in the thesis. A brief introduction to convex polytopes and support functions follows which we use for representing continuous sets. We define the notion of flowpipes and how they are over-approximated with a collection of convex sets in section 2.3. Existing work is revisited on computing the reachable set of hybrid systems having affine continuous dynamics and affine maps on the discrete jumps, with a collection of convex sets. Section 2.3.2 describes the data structure used for storing the flowpipe computed with a support function based reachability algorithm proposed in [GG09]. Section 2.4 presents an introduction to the problem of computing the transition successors in reachability computation. In section 2.4.2, a new approach is proposed for computing the transition successors and it is shown why this new approach should produce more accurate results in theory.

Chapter 3 could be seen as a standalone chapter illustrating an algorithm to compute the support function of the intersection of convex sets with Hyperplanes and Halfspaces efficiently. In the thesis context, this fits in because the novel approach of computing the transition successors in the reachability computation, proposed in section 2.4.2 of the previous chapter, is based on computing efficiently the support function of the intersection of convex sets. It is shown in section 3.1 that the problem of computing the support function of the intersection of a convex set with a hyperplane or halfspace reduces to the problem of minimizing a convex function. Restriction to hyperplane, halfspace set is still worthy because in practice, most of the guard sets in hybrid automata are hyperplanar or halfspace. Furthermore, it is shown that for polyhedral sets, computing the support function of the intersection with hyperplane/halfspace reduces to the minimization of convex piecewise linear function.

Our proposed algorithm for optimizing convex function has two main parts - (1) Minima Bracketing (2) Sandwich algorithm. Minima bracketing is illustrated in section 3.2.1 and the sandwich algorithm is illustrated in section 3.2.2.

In chapter 4, we illustrate the use of our algorithm for computing the support function of the intersection of convex sets with Hyperplanes and Halfspaces, illustrated in chapter 3, in the context of accurate flowpipe-guard intersection. Section 4.1 illustrates the detection of the flowpipe segments which intersect with a given polyhedral guard set. We present the related work in the problem of a convex set and hyperplane intersection in section 4.2.2 and compare our method of computing the support function of the intersection of a polyhedron with hyperplane with the existing work proposed in [GG09].

We also show experiments for computing the support function of the intersection of a polyhedron with a halfspace. Section 4.3 proposes an algorithm to compute the support function of the intersection of a set of convex sets with a hyperplane/halfspace guard set by simultaneously solving a number of minimization problems. Section 4.3.1 proposes an algorithm to compute the convex hull of the intersection of a set of convex sets with a hyperplane/halfspace guard set using branch and bound. Section 4.3.2 shows an algorithm for taking the convex hull of not all but a group of intersection sets between the flowpipe convex sets and the guard set. Section 4.4 shows a way of extending our flowpipe-guard intersection algorithm to polyhedral guard sets. A computational optimization is presented in section 4.5 and the chapter ends by showing the promising results of our flowpipe-guard intersection algorithm on some case studies, namely the Bouncing Ball, the Colliding Pendulums and the Filtered Oscillator. The Navigation benchmark model is also tested.

Chapter 5 presents the SpaceEx tool platform. The requirements analysis and design principles are shown in section 5.1 and section 5.2 respectively. Section 5.3 shows the implementation of two scenarios in the SpaceEx platform, namely the PHAVer and Support Function scenario. The software engineering behind the development of the tool is discussed in section 5.4. The last few sections talks about the input model in SpaceEx, its output formats and about the software licensing.

Chapter 6 is the last chapter of this thesis where the main contributions of this thesis is presented. Some possible directions to future work is also suggested.

Chapter 2

Reachability with Support Functions

This chapter explains the general reachability computation with symbolic states. We define some key terms related to reachability computation like flowpipes, symbolic states and post operators. We present two basic convex set representations, namely convex polytopes and support functions which are used in the symbolic representation of reachable states. The approximation of flowpipes and its computation is explained. In the last section, we discuss the key problem we address in this thesis, i.e., to reduce the approximation error in the computation of transition successors during reachability computation.

Before we talk about what are flowpipes and how is it computed, we first discuss the reachability algorithm. [1]

2.1 Reachability using Symbolic States

Let us recall that an *execution* of the automaton is a sequence of discrete jumps and pieces of continuous trajectories according to its dynamics, and originates in one of the initial states. A state is *reachable* if an execution leads to it. We are concerned with computing the set of states that are reachable and check for *safety*, i.e., given a set of bad states, the reachable set of the system does not intersect with the bad states.

For a set of states R, let the *discrete post-operator* $\mathrm{Post}_d(R)$ be the set of states reachable by a discrete transition from R, and the *continuous post-operator* $\mathrm{post}_c(R)$ be the set of states reachable from R by letting an arbitrary amount of time elapse.

The set of reachable states is the fixpoint of the sequence $R_0 = Init$,

$$R_{k+1} := R_k \cup \mathrm{Post}_d(R_k) \cup \mathrm{post}_c(R_k). \tag{2.1}$$

A straightforward heuristic improvement of this algorithm is to apply both post-operators in alternation, and only to the states new in the previous iteration, leading to Alg. 2.1.

In order to implement Alg. 2.1, we need to efficiently carry out union, difference, and emptiness tests on sets of states, avoiding redundant computations. A common way to do so is to represent sets of states as sets of *symbolic states*[HNSY92]. A symbolic state

[1] This chapter contains excerpts from the publication [FLGD+11] and [FR09].

Algorithm 2.1 Basic Reachability

1: $R, R_N := \text{post}_c(Init)$
2: **while** $R_N \neq \emptyset$ **do**
3: $\quad R' := \text{post}_d(R_N)$
4: $\quad R'' := \text{post}_c(R')$
5: $\quad R_N := R'' \setminus R$
6: $\quad R := R \cup R_N$
7: **end while**

$s = (D, C)$ represents the cross product of a set of discrete states $D \subseteq Loc$ and a set of continuous states $C \subseteq \mathbb{R}^{Var}$. E.g., D could be a single location and C a polyhedron. Let $\mathbb{S}$ be the set of symbolic states of a given hybrid automaton. The post-operators are extended to symbolic states: given a single symbolic state s, $\text{post}_d(s)$ and $\text{post}_c(s)$ both produce a set of symbolic states. The implementation of the verification tool UPPAAL [BLL+96] for real time systems and PHAVer [Fre08] for linear hybrid systems are based on this concept of representing infinite states finitely as symbolic states. The verification tool SpaceEx [FLGD+11] for linear hybrid systems and hybrid systems with affine continuous dynamics which is presented in chapter 5 is also based on representing infinite states finitely as symbolic states.

To represent R and R_N as sets of symbolic states, we use a *passed/waiting list* (PWL), refer to [DBLY02] for a detailed discussion. The passed list contains the symbolic states that have been encountered so far and corresponds in Alg. 2.1 to R. The waiting list contains the symbolic states whose successors still have to be computed. It is implemented as a set of references to elements of the passed list and corresponds in Alg. 2.1 to R_N. The waiting list is computed by performing a set difference on the the newly computed reachable set R'' in each iteration with the passed list ($R'' \setminus R$) which is shown in step 5 of the algorithm. The algorithm terminates when the waiting list is empty. Formally, a PWL is a pair $(P, W) \subseteq 2^{\mathbb{S}} \times 2^{\mathbb{S}}$ with $W \subseteq P$. We define the following operations for the PWL:

- $(P, W) = init(I)$: Assign a set of symbolic states $I \subseteq S$ to P and W.
- $S = diff(s, s')$: Given symbolic states $s = (D, C)$ and $s' = (D', C')$, produces the set of symbolic states $s \backslash s' = \{(D \backslash D', C), (D \cap D', C \backslash C')\}$, or an over-approximation that is efficient to compute. Our default implementation for convex sets C, C' is

$$diff(s, s') = \begin{cases} \{(D \setminus D', C)\} & \text{if } C \subseteq C', \\ \{(D, C)\} & \text{otherwise.} \end{cases}$$

  If C or C' are non convex sets represented as a set of convex sets, we extend this operation pairwise. Let $diff(s, P)$ be the result of applying $diff$ consecutively for all $s' \in P$.
- $(P', W', S) = add(P, W, s)$: Add $s' = diff(s, P)$ to P and W.
- $(P', W') = compact(P, W, S)$: Compact P and W by replacing all $s' \in P, W$ by $diff(s', S)$, eliminating symbolic states where D or C is empty.
- $(s, W') = pop(P, W)$: Select a symbolic state $s \in W$, remove in from W and return it for further processing (post computation).

This leads us to Alg. 2.2, which proceeds as follows:

1. Initialize the PWL with the time-post of the initial states.
2. Pick a symbolic state from the PWL.
3. Apply discrete-post (generating possibly more than one symbolic state).
4. Apply continuous-post to every generated symbolic state.
5. Throw away the symb. states (or parts of them) that are already on the passed list – this involves testing for inclusion and emptiness. Put the remaining ones onto the PWL.
6. Compact the PWL by removing redundant states (this is not always the best reduction; one could also compact first and then add).
7. If the waiting list is not empty, go to 2.

Algorithm 2.2 Reachability using Symbolic States

```
1: (P, W) := init(post_c(Init))
2: while W ≠ ∅ do
3:    s := pop(P, W)
4:    for all s' ∈ post_d(s) do
5:       for all s'' ∈ post_c(s') do
6:          (P, W, S) := add(P, W, s'')
7:          (P, W) := compact(P, W, S)
8:       end for
9:    end for
10: end while
```

The order in which states are popped off the waiting list determines the order of computation (breath first/depth first). This may influence the speed of the computation and may have implications on the interpretation of results. E.g., if a forbidden state is encountered during breath-first exploration, it is the state with the shortest counter example. If over-approximations are used, the resulting set can differ according to which ordering is used, since over-approximation and post operators might not commute.

2.2 Representing Continuous Sets

In n dimensional continuous or hybrid systems, the set of states reachable is a subset of $\mathbb{R}^n$. Representing such continuous sets hence become important. The representation should be such that it is simple and the operations that needs to be done on them by the reachability algorithm should be efficient and tractable. The commonly used representation of continuous sets include Boxes, Ellipsoids, Convex Polytopes, Zonotopes, Simplices and Support function. All the mentioned representations though represent convex sets mainly because of the nice properties of convex sets and the existing mathematical work

on convex analysis. In this thesis, we restrict ourselves to Polytopes and Support function representation of continuous sets.

Restriction to support function and polytopes is justified by the fact that most of the operations that we need to perform on them, for reachable set computation like convex hull, linear transformation, Minkowski sum and intersection are more or less tractable if not efficient. We discuss further about the efficiency of each operation for each of the representation later in this chapter.

In the next section, we present some preliminaries on convex sets and convex functions briefly. We then discuss the two different representations of polytopes followed by the support function representation. A support function uniquely represents a compact convex set or convex bodies. Polyhedral approximation of a compact convex set can be constructed by sampling its support function in a given set of template directions. We lastly present the concept of *flowpipes* to cover the reachable set and its computation with support functions.

2.2.1 Preliminaries

We present here some of the basic definitions and theorems regarding convex sets, convex functions and some operations on sets which we are going to refer later.

The definitions are presented as in [Sch93]. In the notation, by $\ell.x$, we mean the dot product of the vectors ℓ and x where $\ell, x \in \mathbb{R}^n$. $\mathbb{R}^n$ defines a real n-dimensional Euclidean space. sup stands for supremum or least upper bound. $\mathbf{o}$ denotes the zero vector. We say a function $f : \mathbb{R}^n \to \mathbb{R}$ is proper if $\{x \in \mathbb{R}^n | f(x) = -\infty\} = \emptyset$ and $\{x \in \mathbb{R}^n | f(x) = \infty\} \neq \mathbb{R}^n$.

Definition 2.1. A set $\mathcal{X} \subset \mathbb{R}^n$ is convex if together with any two points x, y it contains the segment xy, thus if

$$(1-\lambda)x + \lambda y \in A \text{ for x, y } \in A \text{ and } 0 \leq \lambda \leq 1. \tag{2.2}$$

Definition 2.2. A function $f : \mathbb{R}^n \to \mathbb{R}$ is called *convex* if f is proper and if,

$$f((1-\lambda)x + \lambda y) \leq (1-\lambda)f(x) + \lambda f(y) \tag{2.3}$$

Theorem 2.1. *Let $f_1, \ldots, f_m$ be proper convex functions in $\mathbb{R}^n$, and let*

$$f(x) = \inf \left\{ \sum_{i=1}^{m} f_i(x_i) \;\middle|\; x_i \in \mathbb{R}^n, \sum_{i=1}^{m} x_i = x \right\}. \tag{2.4}$$

Then f is a convex function on $\mathbb{R}^n$. [p-33, [Roc70]]

Definition 2.3. Let $\mathcal{X}$ and $\mathcal{Y}$ be two sets. The Hausdorff distance between $\mathcal{X}$ and $\mathcal{Y}$, denoted $d_H(\mathcal{X}, \mathcal{Y})$ is defined by:

$$d_H(\mathcal{X}, \mathcal{Y}) = max \left(\sup_{x \in \mathcal{X}} \inf_{y \in \mathcal{Y}} \|x - y\|, \sup_{y \in \mathcal{Y}} \inf_{x \in \mathcal{X}} \|x - y\| \right) \tag{2.5}$$

Definition 2.4. The Minkowski sum of two sets $\mathcal{X}$ and $\mathcal{Y}$ is the set of sums of elements from $\mathcal{X}$ and $\mathcal{Y}$:

$$\mathcal{X} \oplus \mathcal{Y} = \{x + y \mid x \in \mathcal{X} \text{ and } y \in \mathcal{Y}\} \tag{2.6}$$

Clearly, $\oplus$ is commutative.

Definition 2.5. For $\mathcal{X} \in \mathbb{R}^n$, the set of all convex combinations of any finitely many elements of $\mathcal{X}$ is called the *convex hull* of $\mathcal{X}$ and is denoted by $\text{CH}(\mathcal{X})$, i.e.,

$$\text{CH}(\mathcal{X}) = \left\{ \sum_{i=1}^{m} \lambda_i x_i \;\middle|\; x_i \in \mathcal{X}, \lambda_i \geq 0, \sum_{i=1}^{m} \lambda_i = 1 \right\} \tag{2.7}$$

Given a matrix $M \in \mathbb{R}^{n \times n}$, $M\mathcal{X} = \{Mx \mid x \in \mathcal{X}\}$ defines the affine image of $\mathcal{X}$.

Proposition 2.1. *For any two closed sets $\mathcal{X}$ and $\mathcal{Y}$, if $\mathcal{B}$ is a ball of radius $d_H(\mathcal{X}, \mathcal{Y})$, then:*

$$\mathcal{X} \subseteq \mathcal{Y} \oplus \mathcal{B} \text{ and } \mathcal{Y} \subseteq \mathcal{X} \oplus \mathcal{B} \tag{2.8}$$

Moreover, $\mathcal{B}$ is the smallest such ball.

It can be seen from the definition of convex sets that intersection, Minkowski sum of convex sets are convex, affine images of convex sets are convex. Also if $\mathcal{X}$ is a convex set, then $\lambda\mathcal{X} = \{\lambda x | x \in \mathcal{X}\}$ is convex.

2.2.2 Convex Polytopes

A convex polytope is a bounded convex polyhedron. A convex polyhedron is the set of points common to one or more half-spaces. A convex polygon is an example of a two dimensional convex polytope.

One way of representing polytopes is as linear constraints, with the interpretation that the polytope is the intersection of the halfspaces that each of the constraint represent,

$$\mathcal{X} = \bigcap_{i=0}^{k} \{x \mid a_i.x \leq \alpha_i\} \tag{2.9}$$

Where $a_i \in \mathbb{R}^n$ and $\alpha_i \in \mathbb{R}$. This is called the H representation of polytopes.

A polytope $\mathcal{X}$ can also be represented by its vertices $x_1, \ldots, x_k$. The polytope $\mathcal{X}$ is then interpreted as the convex hull of its vertices,

$$\mathcal{X} = \left\{ \sum_{i=1}^{k} \lambda_i x_i \mid \lambda_i \geq 0, \sum_{i=1}^{k} \lambda_i = 1 \right\} \tag{2.10}$$

This is called the V representation of polytopes.

Some operations like convex hull are efficient with the V representation and some like the intersection are efficient with the H representation. Conversion between these two representations is a fundamental problem in the theory and application of polyhedra in general. Many algorithms have been proposed for the representation conversion. There is no known approach which efficiently solves the problem in general.

Convex Polytopes is a deeply studied subject in mathematics [Zie95]. The geometry of polyhedra is also interesting in the context of linear programming [DT97], [DT03] since the feasible set defines a convex polyhedra.

There are a number of libraries for the representation and operations on polyhedra like PPL [BHZ08], Polymake [GJ01], CDD [Fuk99] and Polylib [oPFP].

2.2.3 Support Functions

We present the definition of support function and how they represent convex bodies uniquely, followed by some properties of support functions. We then see the support functions of some basic 2-dimensional convex bodies like circle and polygons. We show the graph of support function of some convex polygons in two different domains. The definitions and properties of support function are presented as in [GK98].

Definition 2.6. For a nonempty closed convex set $\mathcal{X} \subset \mathbb{R}^n$ the *support function* $sup_{\mathcal{X}}$ is defined by

$$sup_{\mathcal{X}}(l) = \sup\{\ell.x \mid x \in \mathcal{X}\} \text{ for } \ell \in \mathbb{R}^n. \tag{2.11}$$

Definition 2.7. For a nonempty closed convex set $\mathcal{X} \subset \mathbb{R}^n$ and $\ell \in dom\ sup_{\mathcal{X}}/\{\mathbf{o}\}$, the *supporting plane* $H_{\mathcal{X}}(\ell)$ is defined by

$$H_{\mathcal{X}}(\ell) = \{x \in \mathbb{R}^n \mid \ell.x = sup_{\mathcal{X}}(l)\} \tag{2.12}$$

Similarly, the *supporting halfspace* is defined by

$$H^-_{\mathcal{X}}(\ell) = \{x \in \mathbb{R}^n \mid \ell.x \leq sup_{\mathcal{X}}(l)\} \tag{2.13}$$

Let $\mathcal{S}$ denote a unit sphere in $\mathbb{R}^n$. $sup_{\mathcal{X}}(u)$ is a complete representation of a convex body $\mathcal{X}$, since the values of $sup_{\mathcal{X}}(u)$ for all $u \in \mathcal{S}$ completely defines $\mathcal{X}$, i.e.,

$$\mathcal{X} = \{x \in \mathbb{R}^n \mid x.u \leq sup_{\mathcal{X}}(u) \text{ for all } u \in \mathcal{S}\}. \tag{2.14}$$

This means that $\mathcal{X}$ is the intersection of all the halfspaces $x.u \leq sup_{\mathcal{X}}(u)$.

We state the following result for the characterization of support function.

Proposition 2.2. *Every real-valued function $f(x) : \mathbb{R}^n \to \mathbb{R}$ satisfying the properties:*

1. $f(\mathbf{o}) = \mathbf{0}$
2. $f(\lambda x) = \lambda f(x)$, *for all* $\lambda \geq 0$
3. $f(x + y) \leq f(x) + f(y)$

is a support function of a convex body.

Proposition 2.3. *Every support function $f : \mathbb{R}^n \to \mathbb{R}$ is a convex function.*

Proof.

$$\begin{aligned} f((1-\lambda)x + \lambda y) &\leq f((1-\lambda)x) + f(\lambda y) \text{ [(3) in prop. 2.2]} \\ &\leq (1-\lambda)f(x) + \lambda f(y) \text{ [(2) in prop. 2.2]} \end{aligned}$$

By definition of convex function, f is convex. □

Definition 2.8. Given a compact convex set Ω and directions $\ell_1,\ldots,\ell_r \in \mathbb{R}^n$, the *outer polyhedral approximation* is the polyhedron

$$\lceil\Omega\rceil = \bigcap_{i=1}^{r} \ell_i.x \leq sup_{\Omega}(\ell_i). \tag{2.15}$$

Proposition 2.4. *It holds that* $\Omega \subseteq \lceil\Omega\rceil$. *Moreover, the over-approximation is tight as* Ω *touches the faces of* $\lceil\Omega\rceil$.

It is easy to see that outer polyhedral approximation of a closed convex set can be derived from its support function by sampling it in the directions of interest and taking the intersection of the supporting hyperplanes. We shall see in section 2.3.2 how this property of support functions is used in the flowpipe computation.

Definition 2.9. Given convex sets $\mathcal{S}_1,\ldots,\mathcal{S}_n$ and a set of directions D called template directions, the *template hull* of $\mathcal{S}_1,\ldots,\mathcal{S}_n$ in the template directions is defined as:

$$\mathrm{TH}_D\left(\mathcal{S}_1,\ldots,S_n\right) = \bigcap_{\ell\in D}\{x\in\mathbb{R}^n \mid \ell.x \leq \max_{i\in[1,n]}\left(sup_{\mathcal{S}_i}(\ell)\right)\}. \tag{2.16}$$

It is easy to see that *template hull* of convex sets is simply the *template polyhedron*[SSM05], [SDI08] of the union of the convex sets in the template directions.

We now state some well-known properties of support function:

$$sup_{\mathrm{CH}(\mathcal{X}_1\cup\mathcal{X}_2)}(\ell) = \max\left(sup_{\mathcal{X}_1}(\ell), sup_{\mathcal{X}_2}(\ell)\right), \tag{2.17}$$

$$sup_{M\mathcal{X}}(\ell) = sup_{\mathcal{X}}(M\ell), \tag{2.18}$$

$$sup_{\mathcal{X}_1\oplus\mathcal{X}_2}(\ell) = sup_{\mathcal{X}_1}(\ell) + sup_{\mathcal{X}_2}(\ell). \tag{2.19}$$

To understand a relation between the shape of convex sets and their corresponding support function, we consider some simple convex bodies in $\mathbb{R}^2$ and plot their support function in the *polar domain* as well as in the λ *domain*. An angle θ in the polar domain defines a direction vector $\ell = (cos(\theta), sin(\theta)) \in \mathbb{R}^2$. We need to define what we mean by the λ *domain*. Given two vectors $\ell, n \in \mathbb{R}^n$, $\ell - \lambda n$ for $\lambda \in [-\infty, +\infty]$ spans all directions from vector n to $-n$ in the halfspace containing ℓ (Figure 2.1). The λ domain is important for us when we see later the support function of the intersection of convex sets with hyperplanes and halfspaces.

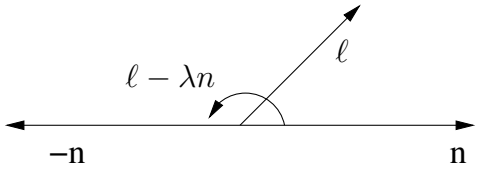
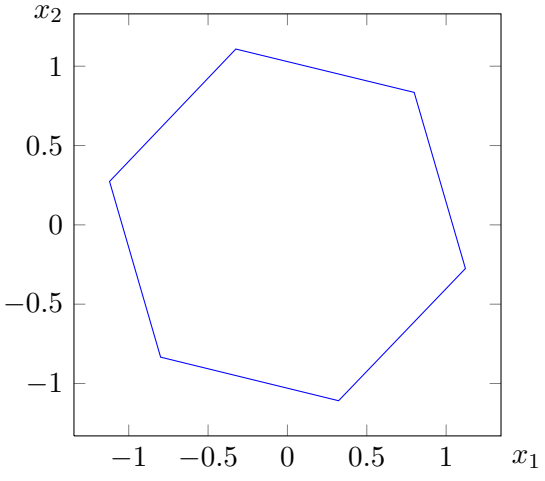


Figure 2.1: $\ell - \lambda n$ denotes all directions from n to -n in the halfspace containing ℓ for $\lambda \in [-\infty, \infty]$

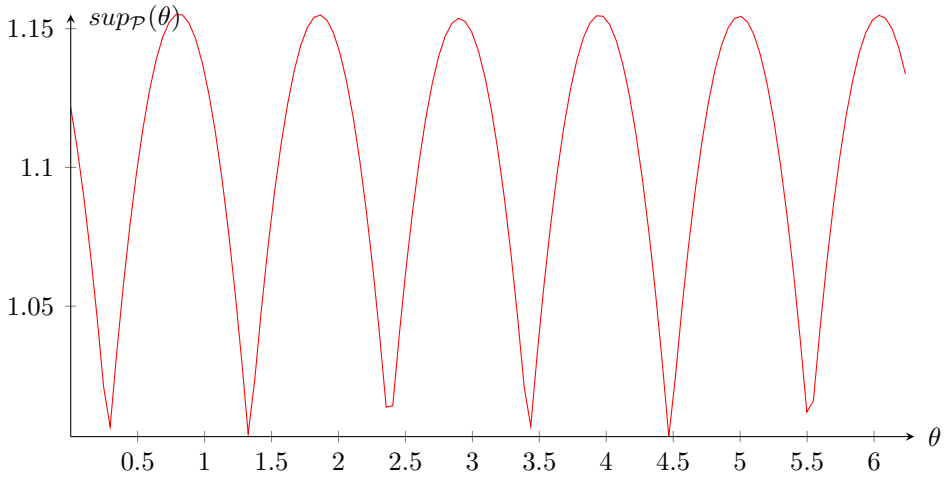
For a circle $C(0, r)$, centered at the origin and with radius r, the support function for any direction vector ℓ is a constant given by $r\|\ell\|$. However, for a circle C centered at (a, b) having radius r, the support function is not a constant and is given by $sup_C(\ell) =$

$r + a\|\ell\|cos(\theta) + b\|\ell\|sin(\theta)$ where $(\|\ell\|cos(\theta), \|\ell\|sin(\theta))$ defines the polar coordinates of ℓ.

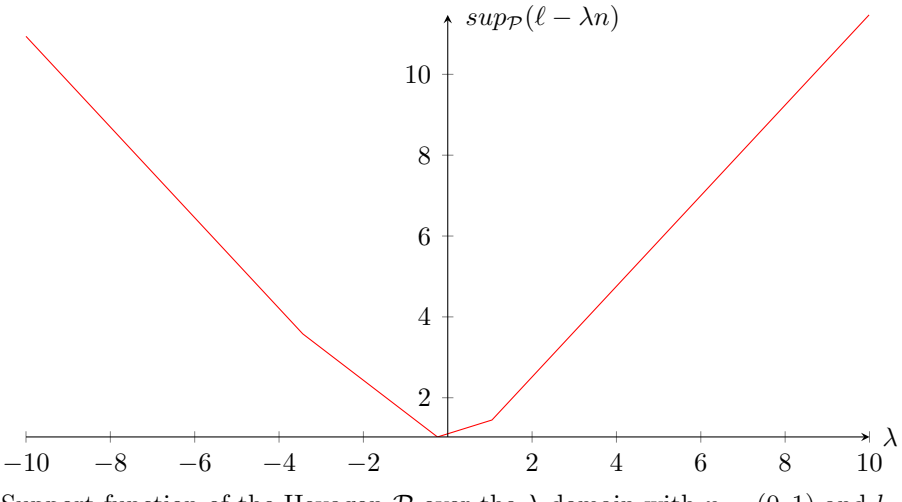
For reachability analysis purpose, we are mainly interested in the support function of polytopes. Let us plot the support function of some regular polygons with varying number of faces both in the polar domain and in the λ domain. Figure 2.2 and Figure 2.3 shows the support function graphs in the polar and λ domain of a hexagon and a polytope with 15 facets respectively. We see that the support function of polytopes in the polar domain are piecewise concave function with number of concave pieces $\approx$ number of faces whereas the support function of polytopes in the λ domain are convex and piecewise linear. Also, observe that with larger number of facets, the support function in the λ domain becomes more and more smooth near the global minima. The reader is referred to [GK98] for a more detailed analysis of convex bodies and their support functions.



(a) A Hexagon $\mathcal{P}$

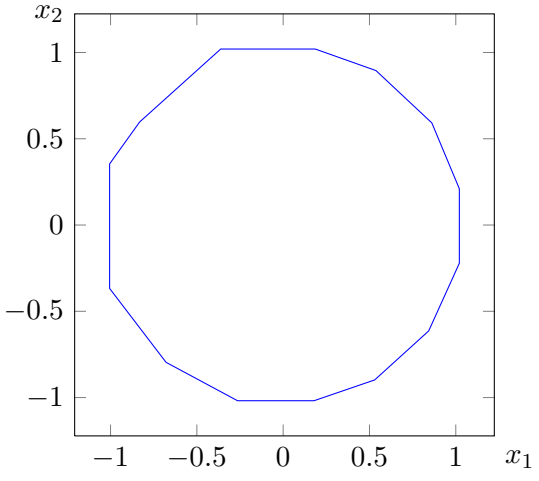


(b) The Support function of the Hexagon $\mathcal{P}$ over the polar domain in $(0, 2\pi)$

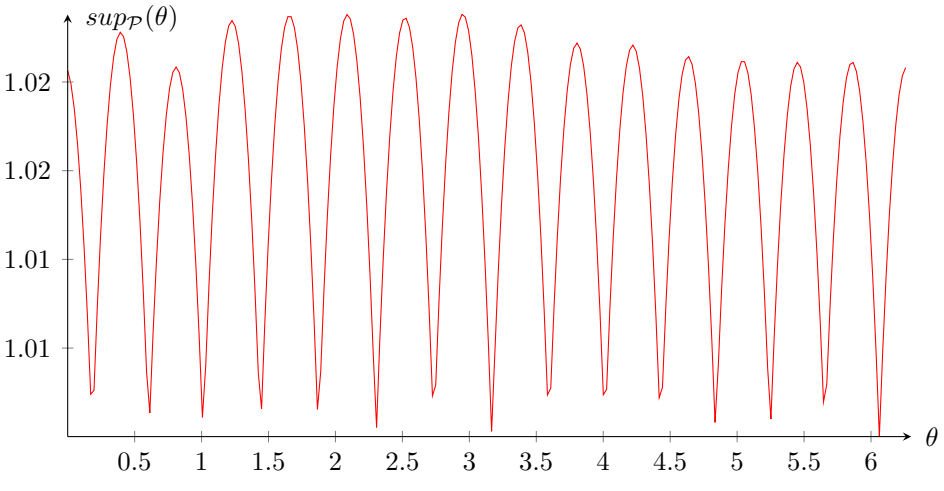


(c) The Support function of the Hexagon $\mathcal{P}$ over the λ domain with $n = (0, 1)$ and $l = (1, 0)$.)

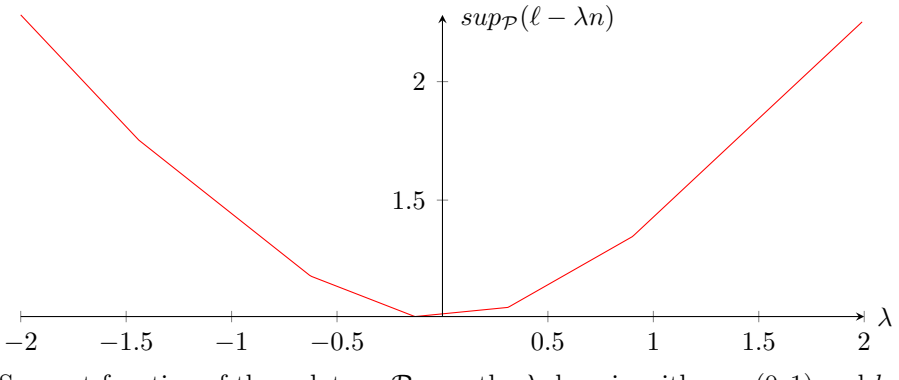
Figure 2.2: The support function of a hexagon in the polar domain and in the λ domain.



(a) A polytope $\mathcal{P}$ with 15 facets



(b) The Support function of the polytope $\mathcal{P}$ over the polar domain in $(0, 2\pi)$

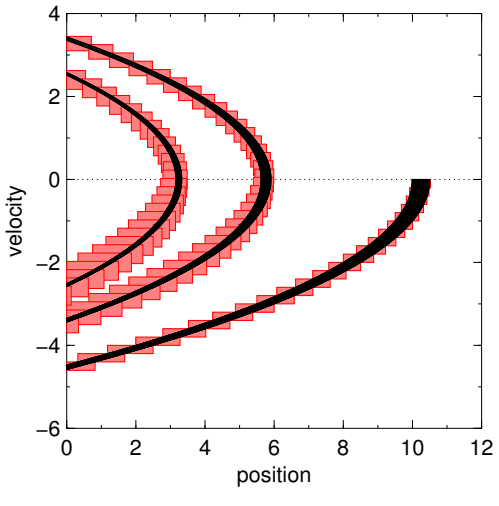


(c) The Support function of the polytope $\mathcal{P}$ over the λ domain with $n = (0, 1)$ and $l = (1, 0)$.)

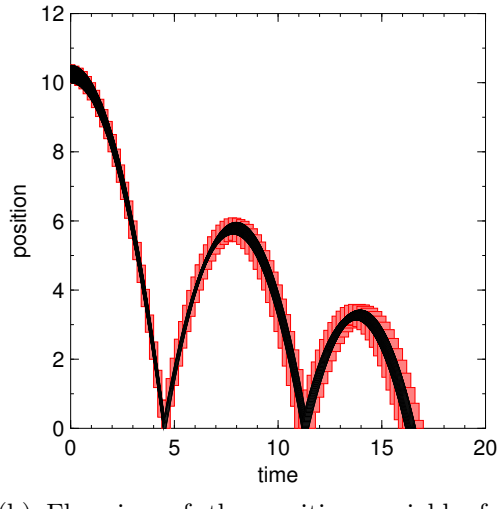
Figure 2.3: The support function of a polytope with 15 facets in the polar domain and in the λ domain. In the λ domain, it can be observed that more the number of facets of the polytope, flatter is the support function near the global minima.

2.3 Computing Time Elapse Successors

We defined reachable set in section 1.4 as the set of all trajectories of the hybrid automata starting from the initial states. A *flowpipe* is the reachable set over an interval of time $[0,t_f]$. The term *flowpipe* is borrowed from the literature [CK98]. Flowpipes can be non-convex but they are approximated by a union of convex sets and that is what we do. We approximate the flowpipe as a union of a finite number of convex sets. Each such convex set is called a flowpipe segment. We define the k^{th} segment of a flowpipe as the convex set which approximates $\mathcal{R}[t_{k-1}, t_k]$. Given a global time horizon T and a discretization time step δ, a flowpipe is approximated with $N = T/\delta$ number of convex segments. We shall refer to such sequence of convex sets that approximates the flowpipe as Ω_i. Figure 2.4(a) shows the flowpipes over the state space and its approximation with the union of convex sets for up to three jumps of the bouncing ball. Figure 2.4(b) shows the flowpipes of the position variable x over time and its approximation.



(a) Flowpipe of the bouncing ball model for three jumps of the ball is shown in black. The approximation of the flowpipe as segments of convex sets is shown in Grey.



(b) Flowpipe of the position variable for three jumps of the ball is shown in black over time. The approximation of the flowpipe as segments of convex sets is shown in Grey.

Figure 2.4: Flowpipe of the Bouncing Ball Model.

2.3.1 Flowpipe Approximation

In this thesis, we consider $Flow(l)$ to be a continuous dynamics of the form

$$\dot{x}(t) = Ax(t) + u(t), \qquad u(t) \in \mathcal{U}, \tag{2.20}$$

where $x(t) \in \mathbb{R}^n$, A is a real-valued $n \times n$ matrix and $\mathcal{U} \subseteq \mathbb{R}^n$ is a closed and bounded convex set. In this section, we discuss how we compute the flowpipe approximation for such dynamics. Let $\mathcal{X}_0$ denote the initial set and we assume it to be a closed convex set. We over-approximate the *flowpipe* by a sequence of continuous sets $\Omega_0, \ldots, \Omega_{N-1}$ that covers the reachable states up to time T (N depends on the chosen time step). Let us first consider the simpler case of *linear time invariant systems*:

$$\dot{x}(t) = Ax(t) \tag{2.21}$$

The analytic solution in this case is given by:

$$\dot{x}(t) = e^{At}x_0 \tag{2.22}$$

Taking this to the set based representation, it is easy to see that $\mathcal{R}(t,t) = e^{At}X_0$, where $\mathcal{R}(t,t)$ denotes the set of states reached at time t. The flowpipe segments Ω_i can be computed using the following recurrence relation:

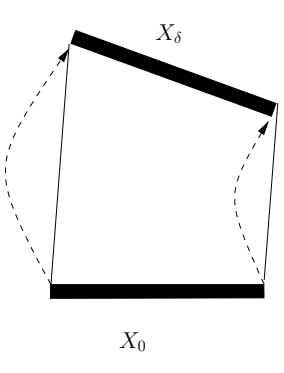
$$\Omega_{i+1} = e^{At}\Omega_i \tag{2.23}$$

Where each Ω_i is such that $\Omega_i \subseteq \mathcal{R}(i\delta, (i+1)\delta)$, δ being the time step.

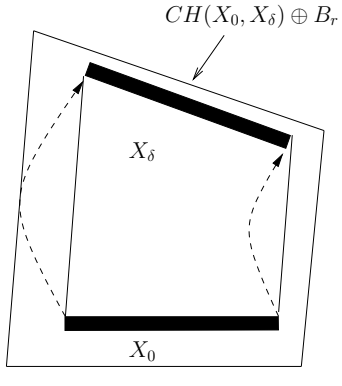
Let us see the computation of Ω_0 which over-approximates the reachable set $\mathcal{R}(0,\delta)$. Let $\Phi = e^{A\delta}$. If $\mathcal{X}_0$ denotes the initial set, $\mathcal{X}_\delta = \Phi\mathcal{X}_0$ denotes the set of states reachable after δ time. We take the convex hull of $\mathcal{X}_0 \cup \mathcal{X}_\delta$ denoting it by $\mathrm{CH}(\mathcal{X}_0, \mathcal{X}_\delta)$. Notice that the convex hull might not enclose the entire flowpipe section as illustrated in Figure 2.5(a). To completely cover the flowpipe segment, a ball $\mathcal{B}$ of sufficient radius is Minkowski sum-ed with the convex hulled set as shown in Figure 2.5(b). Hence we have:

$$\Omega_0 = \mathrm{CH}(\mathcal{X}_0, \mathcal{X}_\delta) \oplus \mathcal{B}_r \tag{2.24}$$

Where $\mathcal{B}_r$ denotes a ball of radius r. Proposition 2.1 guides us to choose the radius to be the Hausdorff distance between exact flowpipe segment and the convex hulled set $\mathrm{CH}(\mathcal{X}_0, \mathcal{X}_\delta)$. This guarantees that the bloated set contains the flowpipe segment completely. A computation of the upper bound on the Hausdorff distance between $\mathcal{R}(0,\delta)$ and $\mathrm{CH}(\mathcal{X}, \mathcal{X}_\delta)$ is shown in [Gir04].



(a) The convex hull set shown with black boundary do not enclose the flowpipe shown with dashed boundary.



(b) The bloated convex hull set encloses the flowpipe.

Figure 2.5: Illustrating the computation of the first flowpipe segment approximation.

Once this over-approximate set Ω_0 is computed, the flowpipe sequence can be computed by applying a linear transformation $\phi = e^{A\delta}$ as shown in (2.23), i.e., $\Omega_{i+1} = \Phi\Omega_i$. Both polyhedra and convex sets in support function representation are closed under linear

transformation and can be used for representing Ω_i. [CK98] shows a method for polyhedral approximation of flowpipes which uses simulation to find the support functions of flowpipe segments in given directions. They also propose a way of estimating directions that constitute the facets of the polyhedra approximating the flowpipe segments.

Let us now consider the more general case of (2.20). The difference here is the additional input term $u(t) \in \mathcal{U}$ in the dynamics which brings in non-determinism. The solution for (2.20) is of the following form:

$$x(t) = e^{tA}x_0 + \int_0^t e^{(t-s)A}u(s)\,ds \tag{2.25}$$

The evolution of the variables can be seen as the superposition of two separate evolutions with time, one with the initial set $\mathcal{X}_0$ and no input set and the other with the initial set as $\{0\}$ and input set $\mathcal{U}$. The first one constitutes the term $e^{tA}x_0$ and the other constitutes the term $\int_0^t e^{(t-s)A}u(s)\,ds$. Let us say that $\int_0^t e^{(t-s)A}u(s)\,ds = \mathcal{R}(t,t)(\{0\})$. We can then decompose the reachable set at the δ time instant $\mathcal{R}(\delta,\delta)$ as:

$$\mathcal{R}(\delta,\delta) = e^{\delta A}\mathcal{X}_0 \oplus R(\delta,\delta)(\{0\}). \tag{2.26}$$

and the reachable set after δ time as:

$$\mathcal{R}(0,\delta) = \bigcup_{t\in[0,\delta]} \left(e^{tA}\mathcal{X}_0 \oplus R(t,t)(\{0\})\right) \tag{2.27}$$

As in the computation of flowpipes for LTI systems, we compute a sequence of Ω_i which covers the reachable set. We compute an over-approximation Ω_0 of the first flowpipe segment such that $\Omega_0 \subseteq \mathcal{R}(0,\delta)$. As we did with LTI systems, we take the convex hull of $\mathcal{X}_0$ and $\Phi\mathcal{X}_0$ and bloat this with a ball of sufficient radius. This time, this ball should consider not only for the curvatures but also for the set of states reachable under the input set, i.e., $R(\delta,\delta)(\{0\})$. Thus, we have $\Omega_0 = \mathrm{CH}(\mathcal{X}_0, \Phi\mathcal{X}_0) \oplus B_\alpha$. The derivation of α is shown in [Gir05]. [Gir05] also shows that a ball of sufficient radius r' can over-approximate $R(\delta,\delta)(\{0\})$, i.e., $R(\delta,\delta)(\{0\}) \subseteq \mathcal{V} = \mathcal{B}_{r'}$. The sequence of Ω_i is then computed using the following recurrence relation:

$$\Omega_{i+1} = \Phi\Omega_i \oplus \mathcal{V}. \tag{2.28}$$

It is also shown in [Gir05] that the Hausdorff distance between the exact flowpipe and its approximation with $\bigcup_{i=0}^{N-1}\Omega_i$ vanishes as the time step δ tends to 0.

There are choices of representing the Ω_i as polytopes, ellipsoids, boxes, zonotopes or convex sets represented by support functions. All we need to consider in the choice is that the set representation should be closed under convex hull, Minkowski sum and linear transformation operation. In this thesis, we shall consider the support function based algorithm proposed by [GG09]. It is shown to be scalable and can analyze affine systems having more than 100 continuous variables. Support function representation of convex sets are efficient with the convex hull, Minkowski sum and linear transformation operations as shown in the properties of support functions in section 2.2.3. In the text which follows, when we refer to flowpipes we mean the sequence Ω_i which approximates the flowpipe unless we explicitly state that it refers to the exact flowpipe.

Flowpipe computation can be seen to consist of two main operations. One is computing the flowpipe inside a location of the hybrid automata for a given dynamics. This we refer

as the post_c operation standing for post continuous. The second operation is computing the map of the flowpipe when there is a discrete jump from one location to another in the hybrid automata. This we refer as the post_d operation standing for discrete post. Let us visit each operation in detail.

2.3.2 Computing Flowpipes with Support Functions

We refer the set of reachable states in a location of the hybrid automata by a location flowpipe. The post_c operation is responsible for this computation. Briefly, given a finite set of directions, post_c computes the support function samples of the time elapse set at N instances, each after δ time in the given directions respectively. δ and N are parameters which need to be carefully chosen for a system, either by the verification tool automatically or supplied by the user. For example, for systems with monotonic behavior over time, δ could be chosen to be large and it could be chosen small for systems which are non-monotonic. In terms of implementation, we need a suitable data structure that stores the flowpipe approximation or the Ω_i sequence. We use a matrix representation which we name as *Support Function Matrix* (SFM). SFM stores the N support function values for each direction defined as follows:

Definition 2.10. Given a set of r directions $L = \{l_1, \ldots, l_r\}$ and a time horizon N, $\overline{\Omega}_1, \ldots, \overline{\Omega}_N$ is represented as a $r \times N$ matrix called *Support Function Matrix* with $(i,j)^{th}$ entry denoting the support function of Ω_j in the direction ℓ_i. For a given SFM M and directions L, we denote the outer polyhedral approximation of the jth set as

$$PO(L, M_j) = \bigcap_{i=1}^{r} \ell_i.x \leq M_{i,j}.$$

Similarly, a $r \times N$ *Support Vector Matrix* (SVM) represents $\underline{\Omega}_1, \ldots, \underline{\Omega}_N$ with the $(i,j)^{th}$ entry denoting the support vector of Ω_j in the direction ℓ_i. The convex hull of the support vectors in the j^{th} column of a SVM will define an under-approximation $\underline{\Omega_j}$ of the flowpipe.

The reachable continuous set resulting from time elapse in a location, say d, is now represented in the form of a SFM.

Example 2.1. *Figure 2.6 shows the polyhedral overapproximation obtained from using our support function implementation of* Post_c *and the dynamics*

$$\begin{aligned} \dot{x} &= -1.3863x + 0.6931y, \\ \dot{y} &= -0.6931y. \end{aligned}$$

For comparison, the Ω_k are shown as outlines. The set of directions for computing the SFM was chosen to be the axis directions.

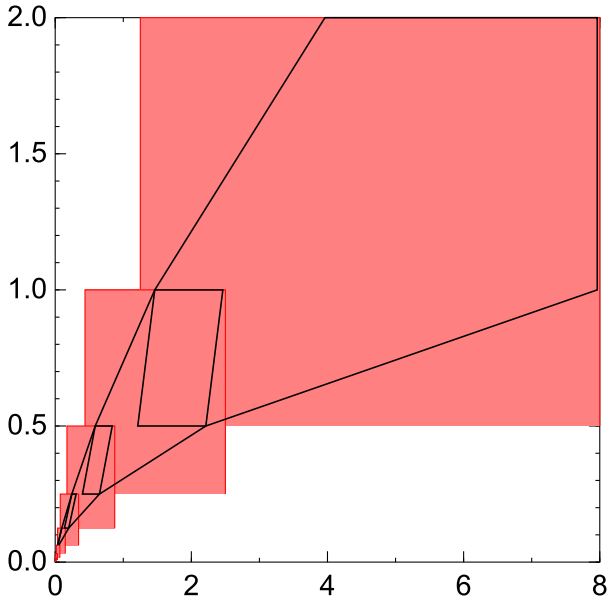


Figure 2.6: Polyhedral overapproximation of Post_c using support functions (shaded), and actual Ω_k for comparison (outlined)

It is to be noticed that the set of directions in the outer polyhedra computation is fixed a priori in the $post_c$ operation. We also call this set the *template directions*. Let us remark that as long we remain with the support function representation of the convex sets Ω_i, we have the complete information about them. Once the template directions are fixed and we compute the outer polyhedra covering the flowpipe, we loose information about the supporting hyperplanes of the other directions. This conversion to polytopes is nevertheless inevitable due to the need of visualization. We cannot graphically plot an abstract functional representation of a set. Another reason for such a conversion is that some operations on support function is not cheap like the intersection and containment operation required for the reachability computation. Intersection operation is cheap for polyhedra in constraint representation and similarly containment check is also cheap for constraint polyhedra and even cheaper if the polyhedra have the same facet normals.

We currently consider the following choices of template directions for an n-dimensional system:

- *box* directions, i.e., $2n$ directions aligned with the axes, i.e., $x_i = \pm 1$, $x_k = 0$ for $k \neq i$;
- *octagonal* directions, i.e., $2n^2$ directions, consisting of all combinations of $x_i = \pm 1$, $x_j = \pm 1$, $x_k = 0$ for $k \neq i, j$;
- *uniform* directions, i.e., a set of m directions that are distributed as uniformly as possible;
- *user-defined* directions, which can be combined with the other types.

However, the algorithm supports a more general choice of directions, which remains to be investigated.

We also need to take into consideration the invariant of the location when computing the flowpipe approximation Ω_i. In our implementation, we test at the k-th step whether Ω_k is entirely outside of the invariant, and stop the sequence once this is the case. Then we intersect the invariant with the computed Ω_k. Note that this procedure may produce an over-approximation, as this procedure of invariant intersection may eliminate some of the trajectories starting in the *Init* states without eliminating all of them. We include the invariant face normals in the template directions, so the result is usually of satisfactory precision.

2.4 Computing Transition Successors

Each flowpipe that is created by the time elapse step is passed to the computation of transition successors. States that take the transition must satisfy the guard, are then mapped according to the assignment and the result must satisfy the invariant of the target location. Let $\mathcal{G}$ be the guard set of the transition, $\mathcal{I}^+$ the invariant of the target location, and let the transition assignment be (1.2). The image of a set $\mathcal{X}$ with respect to the transition is

$$\text{post}_d(\mathcal{X}) = \big(R\big(\mathcal{X} \cap \mathcal{G}\big) \oplus \mathcal{W}\big) \cap \mathcal{I}^+. \tag{2.29}$$

We can see that there are two operations involved, intersection and assignment. We discuss each of the two operation below.

Computing the one-to-one image of the sets covering the flowpipe, as in (2.28), can have the devastating effect of increasing the number of convex sets exponentially with the search depth. To avoid an explosion in the number of sets and gain efficiency, we compute the convex hull or template hull of subsets of these sets instead. This is referred to as *clustering* which is also explained below in detail.

Flowpipe-Guard Intersection As the flowpipe is approximated with a set of convex sets, there can be a set of convex sets intersecting with the guard set. Since intersection with support function represented convex sets is hard, [FLGD+11] computes intersection on the template hull [definition 2.9] of each convex set, i.e., its outer polyhedral approximation in the template directions, say $\mathcal{P}_D = \text{TH}_D(\mathcal{X})$.

The difficulty with intersection is that we do not have an a-priori bound on how the over-approximation error from the flowpipe computation affects the result of the intersection. Moreover, considering the outer polyhedral approximation of the support function represented convex sets adds to the over-approximation. If $\mathcal{G}$ is a polyhedron in constraint form whose constraint normals are included in template directions, then the intersection operation can be carried out very efficiently by taking the minimum of the template coefficients:

$$\mathcal{P}_D \cap \mathcal{G} = \big\{x \in \mathbb{R}^n \mid \bigcap_{\ell_i \in D} \ell_i \cdot x \leq \min(b_i^{\mathcal{X}}, b_i^{\mathcal{G}})\big\}.$$

If the constraint normals of $\mathcal{G}$ are not template directions, we apply normal intersection for constraint polyhedra, which consists of taking the union of their constraints. Note that

although working with the outer polyhedral sets make the intersection with polyhedral guard in constraint form efficient, this incurs higher over-approximation.

Assignment Recall that according to (1.2) transition assignments are of the form $x' = Rx + w, w \in \mathcal{W}$, where $\mathcal{W} \subseteq \mathbb{R}^n$ is a convex set of non-deterministic inputs. In the general case, the assignment operator is therefore

$$Post_{Asgn}(\mathcal{X}) = R\mathcal{X} \oplus \mathcal{W},$$

and can be computed efficiently using support functions. If the assignment is invertible and deterministic, i.e., R is invertible and $\mathcal{W} = \{w_0\}$ for some constant vector w_0, the exact image can be computed efficiently on the polyhedron.

We consider applying the assignment to a constraint polyhedron

$$\mathcal{P} = \{x \in \mathbb{R}^n \mid A_P x \leq b_P\},$$

where $A_P \in \mathbb{R}^m \times \mathbb{R}^n$ and $b_P \in \mathbb{R}^n$ are the coefficients of the constraints of $\mathcal{P}$. The mapped states are computed exactly by mapping the polyhedron :

$$Post_{Asgn}(\mathcal{P}) = \{x \in \mathbb{R}^n \mid A_P R^{-1} x \leq b_P + A_P R^{-1} w_0\}.$$

Intersection with Target Invariant Depending on the assignment, we need to intersect a support function set or a polyhedron with the target invariant. The intersection of a support function with another set is costly to compute. Since intersection is cheap for polyhedra in constraint form, we compute the template hull of the flowpipe guard intersection set and intersect that instead with the polyhedral invariant in constraint form. This incurs an over-approximation error that may be substantial. It can be reduced by increasing the number of template directions though.

For lack of a better term, we call this the *standard* discrete image operator in this thesis:

$$\text{post}_d(\mathcal{X}) \subseteq \left\lceil R\left(\lceil \mathcal{X} \rceil_L \cap \mathcal{G} \cap \mathcal{I}^-\right) \oplus \mathcal{W} \right\rceil_L \cap \mathcal{I}^+. \tag{2.30}$$

Note that if R is invertible and $\mathcal{W}$ is deterministic (a point), the outermost outer approximation is not necessary since the resulting polyhedron can be computed efficiently with exact methods. Also note that, the intersection with the source invariant is pulled in here which we previously stated to be part of the post_c operation. This is because we want to emphasize that [FLGD+11] computes the intersection of the outer polyhedral approximation of $\mathcal{X}$ with the source invariant I^-. This brings in larger over-approximation in the intersection with guard which as a result brings in larger approximation in the overall reachable set.

2.4.1 Computing Transition Successors with Support Functions

The $post_c$ operator gives us an SFM, a matrix representation of $\overline{\Omega}_0, \ldots, \overline{\Omega}_N$ To compute the intersection of the time elapse set with a guard $\mathcal{G}$, we would like to identify the relevant Ω's to consider for computing the intersection with the guard or in other words, we would like to filter out the irrelevant Ω's before proceeding the intersection computation. By relevant, we mean the ones which can possibly intersect with a guard $\mathcal{G}$. One possibility

Algorithm 2.3 Polytope Intersection over an SFM

Require: The directions set D $= l_1, \ldots, l_r$, a $r \times N$ SFM M and polytope G with k linear constraints $\langle l_1^g, x\rangle \leq c_1, \ldots, \langle l_k^g, x\rangle \leq c_k$.

Ensure: SFM M_I representing $M \cap G$

```
1: Allocate memory for an SFM M_I with r + k rows and N columns.
2: for i ← 1 to r do
3:    for j ← 1 to N do
4:       M_I[i][j] ← M[i][j]
5:    end for
6: end for
7: for i ← 1 to k do
8:    D = D ∪ {l_i^g}
9:    for j ← 1 to N do
10:      M_I[r + i][j] ← c_i
11:   end for
12: end for
```

to identify the relevant Ωs' is to compute their distance from $\mathcal{G}$ and check if it is less than or equal to the diameter of Ω. If the distance is greater than the diameter, we know that $\Omega \cap G = \emptyset$ and hence can discard the Ω. The identified Ω's can be filtered by simply dropping the corresponding columns from SFM, SVM and the result after filtering is a smaller SFM, SVM in terms of columns.

To compute the intersection with G, we manipulate our SFM M so as to get a new SFM M_I representing the intersection set. Semantically interpreting M as an outer polyhedral approximation, the intersection is the SFM given by adding the normal vectors of the constraints (faces) of G to the set of directions, and computing the SFM for the new set of directions.

The complexity of constructing the SFM M_I is $O((r+k)N)$, where r is the number of directions considered to compute SFM during the $post_c$ operation, k is the number of constraints in $\mathcal{G}$ and N is the number of columns in SFM M.

The final part of the $post_d$ operator is to compute the linear transform of the intersection set. Support functions have the convenient property that given compact convex sets $X, D \subseteq \mathbb{R}^n$, a direction $l \in \mathbb{R}^n$, and a $n \times n$ transformation matrix C, $sup_{CX \oplus D}(l) = sup_X(C^T l) + sup_D(l)$. Using this property, we over-approximate the transformed set with an SFM M_T defined by

$$M_{i,j} = sup_{PO(L,M_j)}(C^T l_i) + sup_D(l_i).$$

Algorithm 2.4 shows the construction of M_T from M_I.

Clustering For a flowpipe, $\forall[i_{\min}, i_{\max}] \in \mathcal{I}$ we will have $\Omega_{i_{\min}} \cap \mathcal{G}, \ldots \Omega_{i_{\max}} \cap \mathcal{G}$ as initial sets in the next location. For large intervals, this could lead to considerably large number of initial sets in the next location and furthermore, this increase in the number of initial sets could propagate along as we carry on with the reachable set computation, leading to a drastic slowdown in performance. To avoid this situation, we could consider taking the *HULL* of the convex sets either before or after the intersection with the guard set.

Algorithm 2.4 Linear Transformation over an SFM

Require: The directions set D = $\{l_1, \ldots, l_k\}$, $k \times N$ SFM M_I, n × n matrix A and a $1 \times n$ vector v_b.

Ensure: SFM M_T representing $AM_I + v_b$

```
Allocate memory for a SFM M_T with k rows and N columns.
for i ← 1 to k do
    l'_i ← A^T l_i
end for
for j ← 1 to N do
    Construct a Poyhedron P_j by adding constraints:
    for i ← 1 to k do
        P_j.add_constraint(⟨l_i, x⟩ ≤ M_I[i][j])
    end for
end for
for i ← 1 to k do
    for j ← 1 to N do
        M_T[i][j] ← ρ_{P_j}(l'_i) + ⟨v_b, l_i⟩
    end for
end for
```

To consider a somewhat intermediate approach, we apply what we call *clustering* [FLGD+11]. Given a hull operator, clustering reduces the number of sets by replacing groups of these sets with a single convex set, their hull. We use the following clustering algorithm for a given hull operator *HULL*. Let the width of $P_1, \ldots, P_z$ with respect to a direction $l \in D$ be

$$\delta_{P_1,\ldots,P_z}(l) = \max_{i=1,\ldots,z} \rho(l, P_i) - \min_{i=1,\ldots,z} \rho(l, P_i) \tag{2.31}$$

D is the set of template directions considered in the computation of the flowpipe.

Given $P_1, \ldots, P_z$ and a *clustering factor* of $0 \leq c \leq 1$, the clustering algorithm produces a set of polyhedra $Q_1, \ldots, Q_r$, $r \leq z$, as follows:

1. Let $i = 1$, $r = 1$, $Q_r := P_i$.
2. While $i \leq z$ and $\forall l \in D : \delta_{Q_r,P_i}(l) \leq c\delta_{P_1,\ldots,P_z}(l)$, $Q_r := HULL(Q_r, P_i), i := i + 1$.
3. If $i \leq z$, let $r := r + 1, Q_r := P_i$. Otherwise, stop.

We consider two hull operators: template hull, which is fast but very over-approximate, and convex hull, which is comparatively precise but slower.

2.4.2 Increasing Precision

We propose as a contribution in this thesis an *improved discrete image operator* which aims at increasing the precision over the standard discrete image operator (2.30) by computing instead:

$$\text{post}_d\mathcal{X} \subseteq \left\lceil R\big(\mathcal{X} \cap \mathcal{G} \cap \mathcal{I}^-\big) \oplus \mathcal{W} \right\rfloor_L \cap \mathcal{I}^+. \tag{2.32}$$

Basically, we would like to compute the intersection of the support function represented convex sets instead of their outer polyhedral approximations as in the standard discrete image operator, with the guard set. Figure 2.7 shows how this makes a difference in terms of precision. $\mathcal{G}^*$ in the figure means the conjunction of the guard and invariant constraints. To make this idea practical, we need an efficient way of computing the intersection of a support function represented convex set with the guard set. An efficient computation of the support function of the intersection of a closed convex set with a hyperplane/halfspace or a polyhedral guard set is one of the main contribution of this thesis. It is shown in the introduction to chapter 3 that computing the support function of the intersection of convex sets is a convex function minimization problem. In practice, flowpipes are largely approximated as a collection of polytopes and the guard sets are mostly polyhedra too. With polyhedral sets, (3.6) is a parametric linear program (LP), with λ as parameter, and $f(\lambda)$ is continuous, convex, piecewise linear function. The reader is referred to [DT03] for an introduction to parametric linear programming.

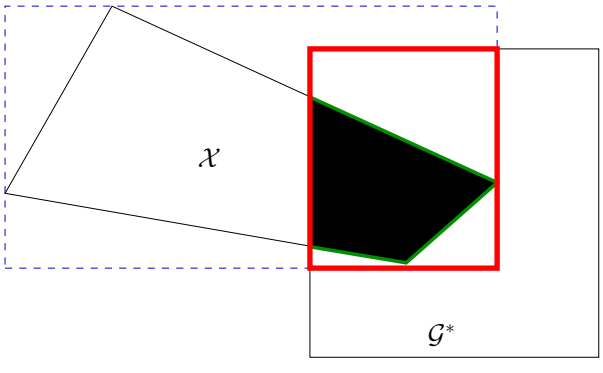


Figure 2.7: Comparing the intersection of the outer polyhedral approximation of set $\mathcal{X}$ and the guard set $\mathcal{G}^*$ (shown in thick bordered region) with the exact intersection (shown in shade)

Another improvement in terms of precision of the discrete image operation is including the pre-image of the target invariant to the intersection step. This can lead to substantial improvements, as shown in Fig. 2.8. Let the target invariant be

$$\mathcal{I}^+ = \left\{ x \;\middle|\; \bigcap_{i=1}^{m} \bar{a}_i^{\mathsf{T}} x \leq \bar{b}_i \right\}.$$

An over-approximation of the pre-image of $\mathcal{I}^+$ with respect to (1.2) is given by

$$\mathcal{I}^* = \left\{ x \;\middle|\; \bigcap_{i=1}^{m} \bar{a}_i^{\mathsf{T}} R x \leq \bar{b}_i + sup_{\mathcal{W}} - \bar{a}_i \right\}. \tag{2.33}$$

Lemma 2.1. $(R\mathcal{X} \oplus \mathcal{W}) \cap \mathcal{I}^+ \subseteq R(\mathcal{X} \cap \mathcal{I}^*) \oplus \mathcal{W}$.
Equality holds if $\mathcal{W} = \{w\}$.

We obtain our image operator

$$\widehat{post}_d(\mathcal{X}) = \left\lceil R\left(\mathcal{X} \cap \mathcal{G} \cap \mathcal{I}^- \cap \mathcal{I}^*\right) \oplus \mathcal{W} \right\rceil_L \cap \mathcal{I}^+. \tag{2.34}$$

With proposition 2.4 and lemma 2.1, it is straightforward to show that this is a tight over-approximation in the following sense:

Lemma 2.2. $\text{post}_d(\mathcal{X}) \subseteq \widehat{post}_d(\mathcal{X})$.
If $\mathcal{W} = \{w\}$, *then* $\widehat{post}_d(()\,\mathcal{X}) = \lceil \text{post}_d(\mathcal{X}) \rceil_L \cap \mathcal{I}^+$.

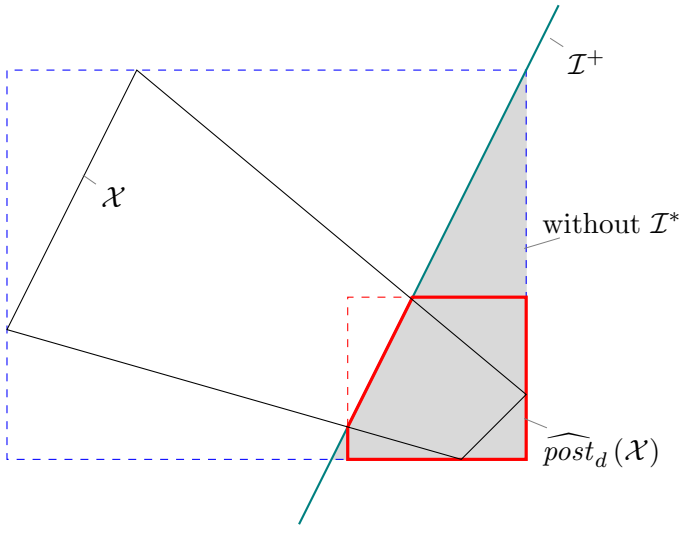


Figure 2.8: The image of $\mathcal{X}$ using the approximation operator (2.34), with the axis directions as template directions. Here, $R = I, \mathcal{W} = 0$, so $\mathcal{I}^+ = \mathcal{I}^*$. $\mathcal{G}$, $\mathcal{I}^-$ are taken to be true. Due to the intersection with the pre-image of the target invariant, $\mathcal{I}^*$, the result of (2.34) (shown in thick red) is considerably more accurate than the same approximation without $\mathcal{I}^*$ (shown shaded gray).

Note that (2.32) includes the source invariant also in the intersection step which is considered to be part of the post_c operation. The reason is because we would like to precisely compute the intersection of support function represented convex sets, we include all such operations together before taking the template hull which then causes loss of accuracy.

$\mathcal{G}, I^-, \mathcal{I}^*$ frequently contain redundant constraints and have matching inequalities that can be simplified to equality constraints. Let $\mathcal{G}^* = \mathcal{G} \cap \mathcal{I}^- \cap \mathcal{I}^*$ be simplified this way. The result of the operator (2.34) is a polyhedral outer approximation. Recalling its definition from (2.15), it involves computing for each $\ell \in L$ the support

$$sup_{R(\mathcal{X} \cap \mathcal{G}^*) \oplus \mathcal{W}}(\ell) = sup_{\mathcal{X} \cap \mathcal{G}^*}(R^\intercal \ell) + sup_{\mathcal{W}}(\ell), \tag{2.35}$$

which we obtain exactly or approximately through minimization as section 4.4 in chapter 4.

In the next chapter we discuss about how we compute the support function of the intersection of a convex set with a hyperplane or halfspace guard set. Chapter 4 then extends this to the precise computation of flowpipe intersection with hyperplanar, halfspace and polyhedral guard sets.

Chapter 3

The Support Function of the Intersection of Convex Sets with Hyperplanes and Halfspaces

In this chapter we explore the problem of computing the support function of the intersection of convex sets with hyperplanes and halfspaces. Unlike other operations like Minkowski sum, linear transformation or convex hull for which support function can be computed efficiently, it is not trivial to compute the support function of the intersection of convex sets. [LG09], [GG09] explores the support function computation of the intersection of convex sets with hyperplanes and shows this to reduce to a unimodal function minimization problem. We follow on the same lines of work proposed there and extend it for halfspace intersection. We show that the support function computation of the intersection of convex sets with hyperplanes and halfspaces reduces to a convex minimization problem and further, for polyhedral sets, to a convex piecewise linear minimization problem. We propose a custom tailored sandwich algorithm to efficiently compute the minima of convex functions which is explained in this chapter.

3.1 Intersection as a Minimization Problem

Let $(\mathcal{X}_i)_{i\in\mathcal{I}}$ (where $\mathcal{I}$ is an arbitrary index set) be a family of non-empty compact convex bodies in $\mathbb{R}^n$ and suppose that their intersection $\mathcal{S}$ is not empty. Then the support function of $\mathcal{S}$ can be represented in the form

$$sup_{\mathcal{S}}(u) = \inf\left\{\sum_{i\in\mathcal{I}} sup_{\mathcal{X}_i}(u_i) \;\middle|\; \sum_{i\in\mathcal{I}} u_i = u\right\}, \tag{3.1}$$

where the infimum is taken over all representations $u = \sum u_i$ with $u_i = \mathbf{o}$ for all but finitely many $i \in I$. [p-46 of [Sch93]]. When only two sets are involved, say $\mathcal{X}$, $\mathcal{Y}$ the above relation can be expressed as follows:

$$sup_{\mathcal{X}\cap\mathcal{Y}}(\ell) = \inf_{w\in\mathbb{R}^d}(sup_{\mathcal{X}}(\ell - w) + sup_{\mathcal{Y}}(w)) \tag{3.2}$$

By proposition 2.3, $sup_{\mathcal{X}}$ and $sup_{\mathcal{Y}}$ is convex and hence by theorem 2.1, $sup_{\mathcal{X}\cap\mathcal{Y}}$ is a convex function.

In the following lemmas concerning the intersection of a compact convex set and a hyperplane or halfspace, the assumption is that their intersection is non-empty because the support function of an empty set is not defined.

Lemma 3.1. *Given a non-empty compact convex set $\mathcal{X}$ and a Hyperplane $\mathcal{H} = \{x : x.n = \gamma\}$, we have*

$$sup_{\mathcal{X}\cap\mathcal{H}}(\ell) = \inf_{\lambda\in\mathbb{R}}(sup_{\mathcal{X}}(\ell - \lambda n) + \lambda\gamma) \tag{3.3}$$

Proof. Any $w \in \mathbb{R}^d$ can be expressed as $w = \lambda_1 n + \lambda_2 n^\perp$, s.t. $\lambda_1, \lambda_2 \in \mathbb{R}$, $n^\perp \in \mathbb{R}^d$ is a unit vector perpendicular to n.

Substituting the above expression of w in (3.2), we get:

$$sup_{\mathcal{X}\cap\mathcal{H}}(\ell) = \inf_{\lambda_1,\lambda_2\in\mathbb{R},n^\perp\in\mathbb{R}^d}(sup_{\mathcal{X}}(\ell - \lambda_1 n - \lambda_2 n^\perp) + sup_{\mathcal{H}}(\lambda_1 n + \lambda_2 n^\perp))$$

For any non-zero λ_2 and $n^\perp$, $sup_{\mathcal{H}}(\lambda_1 n + \lambda_2 n^\perp)$ is ∞.
Since we are interested in finding the infimum, we can restrict ourselves to $\lambda_2 = 0$. Hence, substituting $\lambda_2 = 0$ in the equation and renaming λ_1 to λ, we get :

$$\begin{aligned} sup_{\mathcal{X}\cap\mathcal{H}}(\ell) &= \inf_{\lambda\in\mathbb{R}}(sup_{\mathcal{X}}(\ell - \lambda n) + sup_{\mathcal{H}}(\lambda n)) \\ &= \inf_{\lambda\in\mathbb{R}}(sup_{\mathcal{X}}(\ell - \lambda n) + \lambda\gamma) \end{aligned}$$

Since $\sup_{\mathcal{H}}(\lambda n) = \lambda sup_{\mathcal{H}}(n) = \lambda\gamma$, from the equation of $\mathcal{H} : x.n = \gamma$. □

Lemma 3.2. *Given a non-empty compact convex set $\mathcal{X}$ and a halfspace $\mathcal{H} = \{x : x.n \leq \gamma\}$, $n \in \mathbb{R}^d$ and $\gamma \in \mathbb{R}$*

$$sup_{\mathcal{X}\cap\mathcal{H}}(\ell) = \inf_{\lambda\in\mathbb{R}^+}(sup_{\mathcal{X}}(\ell - \lambda n) + \lambda\gamma) \tag{3.4}$$

Proof. Since $\mathcal{H}$ is bounded only in the direction of vector n, we have:
$\forall w \in \mathbb{R}^d$, if $w = \lambda n$ where $\lambda \in \mathbb{R}^+$ then $sup_{\mathcal{H}}(w) = \lambda\gamma$. $sup_{\mathcal{H}}(w) = \infty$ otherwise.

As we are interested in finding the infimum of the rhs of (3.2), we are only interested in all $w \in \mathbb{R}^d$ s.t $w = \lambda n$, $\lambda \in \mathbb{R}^+$. Substituting for w in (3.2), we have:

$$\begin{aligned} sup_{\mathcal{X}\cap\mathcal{H}}(\ell) &= \inf_{\lambda\in\mathbb{R}^+}(sup_{\mathcal{X}}(\ell - \lambda n) + sup_{\mathcal{H}}(\lambda n)) \\ &= \inf_{\lambda\in\mathbb{R}^+}(sup_{\mathcal{X}}(\ell - \lambda n) + \lambda\gamma) \end{aligned}$$

□

Lemma 3.3. *Consider the halfspace $\mathcal{H} = \{x : x.n \leq \gamma\}$, the hyperplane $\mathcal{H}' = \{x : x.n = \gamma\}$, a compact convex set $\mathcal{X}$, $\ell \in \mathbb{R}^d$ and $\lambda \in \mathbb{R}$ and let*

$$f(\lambda) = sup_{\mathcal{X}}(\ell - \lambda n) + \lambda\gamma \tag{3.5}$$

Then we have

$$sup_{\mathcal{X}\cap\mathcal{H}}(\ell) = \inf_{\lambda\in\mathbb{R}^+} f(\lambda), \quad sup_{\mathcal{X}\cap\mathcal{H}'}(\ell) = \inf_{\lambda\in\mathbb{R}} f(\lambda). \tag{3.6}$$

Proof. The proof follows from the proof of lemma 3.1 and lemma 3.2. □

Let us note some facts about the support function of the intersection of a convex set with a halfspace. We use these facts in our implementation to readily compute the support function and to check for emptiness.

Lemma 3.4. *Let $\mathcal{X}$ be a compact convex set, $\mathcal{H} = \{x \in \mathbb{R}^d : x.n = \gamma\}$ be a hyperplane and $\ell \in \mathbb{R}^d$ s.t. $\ell = \lambda n$, $\lambda \in \mathbb{R}$. If $\mathcal{X} \cap \mathcal{H} \neq \emptyset$, then $sup_{\mathcal{X}\cap\mathcal{H}}(\ell) = \lambda\gamma$.*

Proof. We have,

$$x.n = \gamma, \forall x \in \mathcal{X} \cap \mathcal{H} \tag{3.7}$$

Now,

$$\begin{aligned} sup_{\mathcal{X}\cap\mathcal{H}}(\ell) &= sup_{\mathcal{X}\cap\mathcal{H}}(\lambda n) \\ &= \max_{x\in\mathcal{X}\cap\mathcal{H}}(\lambda n x) \\ &= \lambda \max_{x\in\mathcal{X}\cap\mathcal{H}}(nx) \\ &= \lambda\gamma, \textit{ using } (3.7) \end{aligned}$$

□

Lemma 3.5. *Let $\mathcal{X}$ be a compact convex set and $\mathcal{H} = \{x \in \mathbb{R}^d : n.x \leq \gamma\}$, $n \in \mathbb{R}^d$ and $\gamma \in \mathbb{R}$ be a halfspace. If $sup_{\mathcal{X}}(n) \leq \gamma$, then $\sup_{\mathcal{X}\cap\mathcal{H}}(\ell) = sup_{\mathcal{X}}(\ell)$*

Proof.

$$\begin{aligned} sup_{\mathcal{X}}(n) \leq \gamma &\implies \max_{x\in\mathcal{X}}(n) \leq \gamma. \\ &\implies n.x \leq \gamma, \forall x \in \mathcal{X}. \\ &\implies \mathcal{X} \subset \mathcal{H}. \\ &\implies \mathcal{X} \cap \mathcal{H} = \mathcal{X}. \\ &\implies sup_{\mathcal{X}\cap\mathcal{H}}(\ell) = sup_{\mathcal{X}}(\ell). \end{aligned}$$

□

Lemma 3.6. *Given a compact convex set $\mathcal{X}$ and a halfspace $\mathcal{H}$, we have $f(\lambda) \to -\infty$ as $\lambda \to +\infty \iff \mathcal{X} \cap \mathcal{H} = \emptyset$*

Proof. By support function property, we have

$$\begin{aligned} & sup_{\mathcal{X}}(\ell - \lambda n) \leq sup_X(\ell) + sup_{\mathcal{X}}(-\lambda n) \\ & \implies sup_{\mathcal{X}}(\ell - \lambda n) + \lambda\gamma \leq sup_X(\ell) + sup_{\mathcal{X}}(-\lambda n) + \lambda\gamma \\ & \implies f(\lambda) \leq sup_{\mathcal{X}}(\ell) + \lambda c \end{aligned}$$

Where $c = (sup_{\mathcal{X}}(-n) + \gamma)$. (3.8)

(a) Let $\mathcal{X} \cap \mathcal{H} = \emptyset$. By lemma 4.2, we have

$$\begin{aligned} & - sup_{\mathcal{X}}(-n) > \gamma \\ & \implies sup_{\mathcal{X}}(-n) + \gamma < 0 \\ & \implies c < 0. \end{aligned}$$

In (3.8), $sup_{\mathcal{X}}(\ell)$ is a constant and since $c < 0$, $\lambda \to \infty \implies \lambda c \to -\infty$. Therefore, $f(\lambda) \to -\infty$ as $\lambda \to \infty$.

(b) Let $f(\lambda) \to -\infty$ as $\lambda \to \infty$.

Assume that $\mathcal{X} \cap \mathcal{H} \neq \emptyset$.
From lemma 3.3, we have

$\inf_{\lambda \in \mathbb{R}^+} f(\lambda) = sup_{\mathcal{X} \cap \mathcal{H}}(\ell)$
$\implies \forall \lambda \geq 0, f(\lambda) \geq sup_{\mathcal{X} \cap \mathcal{H}}(\ell)$.

Which contradicts the premise $f(\lambda) \to -\infty$ as $\lambda \to \infty$. Hence, our assumption that $\mathcal{X} \cap \mathcal{H} \neq \emptyset$ must be false. Hence, $\mathcal{X} \cap \mathcal{H} = \emptyset$.
(a) and (b) proves the lemma. □

Lemma 3.7. *Given a compact convex set $\mathcal{X}$ and a halfspace $\mathcal{H}$, if $\mathcal{X} \cap \mathcal{H} \neq \emptyset$, then $f(\lambda) \geq -sup_{\mathcal{X}}(-\ell)$.*

Proof. $\mathcal{X} \cap \mathcal{H} \neq \emptyset$ and $\mathcal{X}$ is compact $\implies \mathcal{X} \cap \mathcal{H}$ is bounded. Hence $sup_{\mathcal{X} \cap \mathcal{H}}(\ell)$ is defined.
Let $x' \in \mathcal{X} \cap \mathcal{H}$ such that $sup_{\mathcal{X} \cap \mathcal{H}}(\ell) = \ell.x'$.
$x' \in \mathcal{X} \cap \mathcal{H} \implies x' \in X$.
Therefore, we have $\ell.x' \geq \min_{x \in X}(\ell.x)$.
$\implies sup_{\mathcal{X} \cap \mathcal{H}}(\ell) \geq -sup_{\mathcal{X}}(-\ell)$.
By lemma 3.3, we have $sup_{\mathcal{X} \cap \mathcal{H}}(\ell) = \inf_{\lambda \in \mathbb{R}^+} f(\lambda)$.
$\implies \inf_{\lambda \in \mathbb{R}^+} f(\lambda) \geq -sup_{\mathcal{X}}(-\ell)$.
$\implies f(\lambda) \geq -sup_{\mathcal{X}}(-\ell)$. □

Lemma 3.8. *$f(\lambda)$ is a convex function.*

Proof. Let $f_1(\lambda) = sup_{\mathcal{X}}(l - \lambda n)$. We know that support function of non-empty compact convex sets are convex functions. Therefore, $f_1(\lambda)$ is convex since $\mathcal{X}$ is a compact convex set.

Let $f_2(\lambda) = \lambda\gamma$, where γ is a constant in $\mathbb{R}$. Since by convention $-\infty < \lambda < \infty$, f_2 is proper. It is hence easy to see that f_2 is convex.

Since pointwise addition of convex functions are also convex, $f(\lambda) = f_1(\lambda) + f_2(\lambda)$ is a convex function. □

When none of the above mentioned special cases could be applied, we compute the support function of the intersection of convex sets with hyperplanes and halfspaces by solving the optimization problem mentioned in lemma 3.1 or lemma 3.2.

3.2 Solving the Minimization Problem

We now present our approach of convex function minimization as a variant of the sandwich algorithm [RR92] in this section. Our algorithm is designed keeping in mind the striking property of convex functions $f(x)$ that if x is a point where $f(x)$ attains a local minima, then $f(x)$ also attains the global minima at x [Roc70]. Convexity also allows us to compute the optimality gap and thus obtain a result of guaranteed accuracy.

Our minimization algorithm is similar to sandwich algorithms used in literature mainly for approximating convex functions with piecewise linear functions, see [BHR91] and references therein, though our focus is to reach the point where the function attains the minima with as few function evaluations as possible. We mention a point say x in the domain of the function where the function f is going to be evaluated as a *sampling point* of the function. Sometimes we also mention a *sample* of the function f by which we mean the pair $(x, f(x))$.

Our proposed algorithm has two parts:

- Minima Bracketing: The minimization algorithm begins with the search of four sampling points, that we call pivots, which bracket the minima. The convex function is evaluated at these pivot points to initialise the iterative convergence algorithm.
- The Sandwich Algorithm : Selecting a new sampling point at each iteration to reduce the optimality gap and continuing the iterations until the minima is precisely reached or the optimality gap is less than or equal to a given bound.

The goal is to find the exact minima or an interval of optimal gap containing the minima with as few function evaluations as possible because in our case evaluating the function is a computationally expensive operation (computing the support function of a compact convex set). Notice that for the initial bracketing of the function minima, we need a minimum of three function evaluations $f(x_1)$, $f(x_2)$ and $f(x_3)$ such that $x_1 < x_2 < x_2$ and $f(x_1) > f(x_2)$, $f(x_2) < f(x_3)$. In our algorithm, we start with four initial evaluations because an extra point aids us to discover tighter **lower bound** on the function minima as is explained in section 3.2.2 illustrating our sandwich algorithm.

We call an interval [a,b] a bracketing interval if $f(a) \leq f(x_{\min}) \leq f(b)$ and $a \leq x_{\min} \leq b$, where $f(x_{\min})$ denotes the function minima, $x_{\min}$ denotes the point in the function domain where the minima is attained and $x_{\min}$, a, $b \in dom(f)$.

3.2.1 Minima Bracketing

Minima bracketing is the process of finding an interval in the domain of the function such that the global minima of the function lies inside that interval. For minima bracketing, we find four pivots namely p_1, p_2, p_3 and p_4 such that the following condition holds:

$$p_1 < p_2 < p_3 < p_4 \tag{3.9}$$

$$f(p_1) > f(p_2) \ \textit{and} \ f(p_3) < f(p_4) \tag{3.10}$$

Convexity of the function shall ensure that $[p_1, p_4]$ is the bracketing interval. We have implemented two algorithms for minima bracketing, one uses *Golden Ratio* and we name it as *golden descent* method. The other is based on *parabolic extrapolation*. These techniques are adapted from [PVTF02] and we have more or less used the routines mentioned there with some modifications. We use the following lemmas in the minima bracketing algorithm:

Lemma 3.9. *Let $x_{\min}$ be a point in the domain of f such that $f(x_{\min}) = f_{\min}$, $f_{\min}$ denoting the function minima. Given two points s, t in the domain of f such that $s < t$, if $f(s) = f(t)$ then $s \leq x_{\min} \leq t$.*

Proof. Consider a point $x \in dom(f)$ such that $x < s$. Let us assume that $f(x) = f_{\min}$ and $f_{\min} < f(s)$
Let $\exists 0 \leq p \leq 1$ such that $p.x + (1-p).t = s$.

By convexity property of f, we know:-

$$\begin{aligned} & p.f_{\min} + (1-p).f(t) \geq f(p.x + (1-p).t) \\ \implies \quad & p.f_{\min} + (1-p).f(s) \geq f(s), \textit{since } f(s) = f(t) \\ & \textit{and } (p.x + (1-p).t) = s \\ \implies \quad & p.f_{\min} \geq p.f(s) \\ \implies \quad & f_{\min} \geq f(s) \end{aligned} \tag{3.11}$$

(3.11) contradicts our assumption that $f(x) = f_{\min}$ and $f_{\min} > f(s)$ Therefore,

$$s \leq x_{\min} \tag{3.12}$$

Similarly, using convexity property of f, we can show that

$$x_{\min} \leq t \tag{3.13}$$

Using (3.12) and (3.13), we know:

$$s \leq x_{\min} \leq t \tag{3.14}$$

□

Corollary 3.1. *Let $x_{\min}$ be a point in the domain of a convex function f such that $f(x_{\min}) = f_{\min}$, $f_{\min}$ denoting the function minima. Given points x_1, x_2 and x_3 in the domain of f,*
If $f(x_1) = f(x_2) = f(x_3)$ then $f_{\min} = f(x_1) = f(x_2) = f(x_3)$. Also $f(x_1)$, $f(x_2)$ and $f(x_3)$ are collinear.

Proof. Using lemma 3.9, we have $f(x_1) \leq f_{\min} \leq f(x_2)$ and $f(x_2) \leq f_{\min} \leq f(x_3)$. Since every local minima of a convex function is a global minima, it must be the case that $f_{\min} = f(x_1) = f(x_2) = f(x_3)$. Also, this means that $f(x_1)$, $f(x_2)$ and $f(x_3)$ are collinear points. □

3.2.1.1 Golden Descent

In *golden descent* approach of minima bracketing, we make an initial guess of two sampling points (In our implementation, we made a choice of sampling the function at 0 and 1). These two sampling points will be our two initial pivot points and we then start decending downhill, increasing the step size by a constant factor until we find the turning point of

the function. We define a turning point to be the first discovered point x in the domain of the function such that $f(x) > f(x_{prev})$ where x_{prev} is the previous sampling point in the decent. We always keep track of the last three samples while descending downhill until we find the turning point of the function. When we discover the turning point for the first time, we assign this as the fourth pivot point p_4 and the third pivot point p_3 is assigned the mid point between p_2 and p_4. This way the pivot conditions in (3.9) and (3.10) hold.

Consider the context of minimizing $f(\lambda)$ of lemma 3.3. For the halfspace intersection case, the search space is in $\mathbb{R}^+$ which means the directions sweep from ℓ to n. Initial sampling at $\lambda = 0$ and $\lambda = 1$ would mean computing the support function of the convex set $\mathcal{X}$ in the direction ℓ and $\ell - n$ respectively which is half the search space apart. The advantage we think of having initial pivots p_1 and p_2 half the search space apart is that during the downhill descend, we reject half the search space at the first go when the new sample is not the turning point. We use the *golden ratio* as the constant to increase the step size at each downhill descend step. The golden ratio is an irrational mathematical constant, approximately 1.61803398874989. The procedure is shown in algorithm 3.1 and the downhill descend is illustrated in Figure 3.1. The reader is referred to [Liv08] for further reading concerning the golden ratio. Notice that for finding the support function of a convex set intersected with a halfspace [lemma 3.2], we need to search the minima in the domain of $\mathbb{R}^+$. In this case we cannot descend at the left of 0. This special case is described in algorithm 3.2. The symmetric case is when the search domain in $-\infty, u$ which is treated by minimizing $f' = -f(x)$.

We might under certain conditions know a priori the domain which contains the function minima, in which case we already have our pivot points p_1 and p_4 and we need to find two more in between them. This case is illustrated in algorithm 3.3. Notice in the algorithm that under some conditions, we end up finding the minima in the process of bracketing.

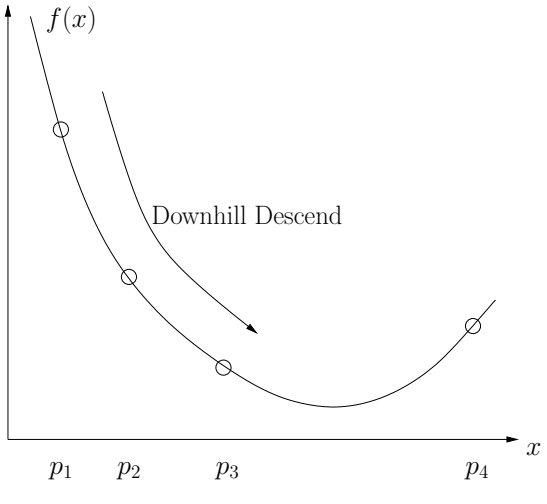


Figure 3.1: Downhill descend with four points. p_4 is the first discovered turning point during the descend.

3.2.1.2 Parabolic Extrapolation

Like in the golden descent method, we make an initial guess of 2 points to sample the function (0 and 1 in our implementation). We then start moving downhill with a step

size by the result of a parabolic extrapolation of the preceeding points that is designed to take us to the extrapolated turning point. Once we find the third point, say p, we move further uphill by 5 to get the fourth pivot point. The difference between the two initial bracketing routine is the choice of the step-size to descend downhill. In golden descent method, the step size is a factor of *golden ratio* whereas in parabolic extrapolation, the step size is guided by the parabolic extrapolation of the minima. For detailed pseudo code refer to [PVTF02].

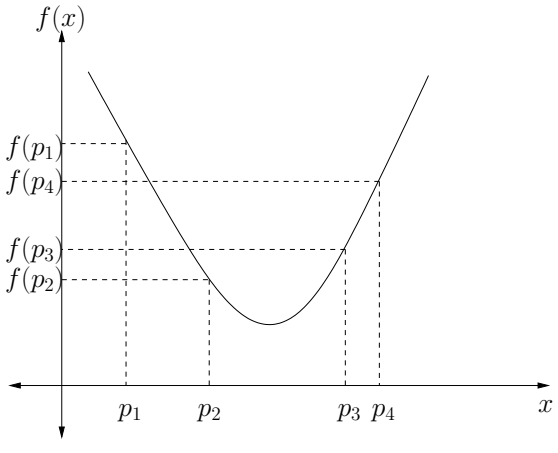


Figure 3.2: Selection of four pivot points p_1, p_2, p_3 and p_4, satisfying the conditions: $p_1 < p_2 < p_3 < p_4$, $f(p_1) > f(p_2)$ and $f(p_3) < f(p_4)$

Algorithm 3.1 Minima bracketing with golden descent method in the domain of $\mathbb{R}$

Require: $p_1 = 0, p_2 = 1$ and $f(p_1), f(p_2) \in \mathbb{R}$.
Ensure: p_1, p_2, p_3, p_4 such that $p_1 < p_2 < p_3 < p_4$ and $f(p_1) > f(p_2)$, $f(p_3) < f(p_4)$.

$GOLD \leftarrow 1.618034$
if $f(p_2) > f(p_1)$ **then**
    swap(p_1,p_2)
end if
$p_3 \leftarrow p_2 + (p_2 - p_1) * GOLD$
while $f(p_3) < f(p_2)$ **do**
    shift2(p_1, p_2, p_3) {Shifts p_1 to p_2, p_2 to p_3.}
    $p_3 \leftarrow p_2 + (p_2 - p_1) * GOLD$
end while
if $p_1 > p_2$ **then**
    swap(p_1, p_3) {Rename the pivots to increasing order}
end if
$p_4 = p_3$
$p_3 = p_2 + (p_4 - p_2)/2$

Algorithm 3.2 Golden descent with search range as l to $+\infty$

Require: $s_1 = l, s_2 = l + 1$ and $f(s_1), f(s_2) \in \mathbb{R}$.
Ensure: p_1, p_2, p_3, p_4 such that $p_1 < p_2 < p_3 < p_4$ and $f(p_1) > f(p_2)$ and $f(p_3) < f(p_4)$

$GOLD \leftarrow 1.618034$
if $f(s_2) < f(s_1)$ **then**
  $p_1 \leftarrow s_1$ and $p_2 \leftarrow s_2$
  $p_3 \leftarrow p_2 + (p_2 - p_1) * GOLD$
  while $f(p_3) < f(p_2)$ **do**
    shift2(p_1, p_2, p_3) {Shifts p_1 to p_2 and p_2 to p_3.}
    $p_3 \leftarrow p_2 + (p_2 - p_1) * GOLD$
  end while
  $p_4 \leftarrow p_3$
  $p_3 \leftarrow p_2 + (p_4 - p_2)/2$
else
  $p_1 \leftarrow s_1$ and $p_4 \leftarrow s_2$
  $p_3 \leftarrow p_1 + (p_4 - p_1)/2$ {Move towards p_1 until sample is less than $f(p_1)$}
  $s \leftarrow p_1 + (p_3 - p_1)/2$
  $s_{prev} \leftarrow p_3$
  while $f(p_1) \leq f(s)$ **do**
    if p_1, s, s_{prev} are collinear **then**
      $p_4 \leftarrow p_3 \leftarrow p_2 \leftarrow p_1$ {We conclude the minima to be $f(p_1)$} {All pivots are set to p_1}
      stop
    end if
    $s_{prev} = s$
    $s \leftarrow p_1 + (s - p_1)/2$
  end while
  $p_2 \leftarrow s$
end if

Algorithm 3.3 Pivot selection with the search range as l to u

Require: $s_1 = l, s_2 = u$ and $f(s_1), f(s_2) \in \mathbb{R}$.
Ensure: p_1, p_2, p_3, p_4 such that $p_1 < p_2 < p_3 < p_4$ and $f(p_1) > f(p_2)$,$f(p_3) < f(p_4)$

$p_1 \leftarrow s_1$ and $p_4 \leftarrow s_2$ {Min lies between $f(p_1)$ and $f(p_4)$}
if $f(s_1) > f(s_2)$ **then**
  $p_2 \leftarrow p_1 + (p_4 - p_1)/2$
  $s \leftarrow p_2 + (p_4 - p_2)/2$
  $s_{prev} \leftarrow p_2$ {Now, Keep going right until sample is less than $f(p_4)$ sample}
  while $f(p_4) \leq f(s)$ **do**
    if p_4, s, s_{prev} are collinear **then**
      $p_1 \leftarrow p_2 \leftarrow p_3 \leftarrow p_4$ {We conclude the minima to be $f(p_4)$} {All pivots are set to p_4}
      stop
    end if
    $s_{prev} = s$
    $s \leftarrow s + (p_4 - s)/2$
  end while
  $p_3 = s$
else if $f(s_1) < f(s_2)$ **then**
  $p_3 \leftarrow p_1 + (p_4 - p_1)/2$ {Move towards p_1 until sample is less than $f(p_1)$}
  $s \leftarrow p_1 + (p_3 - p_1)/2$
  $s_{prev} \leftarrow p_3$
  while $f(p_1) \leq f(s)$ **do**
    if p_1, s, s_{prev} are collinear **then**
      $p_4 \leftarrow p_3 \leftarrow p_2 \leftarrow p_1$ {We conclude the minima to be $f(p_1)$} {All pivots are set to p_1}
      stop
    end if
    $s_{prev} = s$
    $s \leftarrow p_1 + (s - p_1)/2$
  end while
  $p_2 \leftarrow s$
else
  $p_2 = p_1 + (p_4 - p_1)/3$ and $p_3 = p_1 + 2 * (p_4 - p_1)/3$
  if $f(p_2) = f(p_4)$ or $f(p_3) = f(p_4)$ **then**
    $p_1 \leftarrow p_2 \leftarrow p_3 \leftarrow p_4$ {We conclude that the function plot between p_1 and p_4 is a straight line and the minima is the function sample at any point between p_1, p_4.}
    stop
  end if
end if

3.2.2 A Sandwich Algorithm for the Direct Minimization of Convex Functions

In this section, we describe our sandwich algorithm to find the minima of a convex function of one variable given a minima bracketing. We describe the algorithm for general convex functions. We then show the performance of the procedure for convex piecewise linear functions which is a special case of convex functions. The procedure is inspired from the

sandwich algorithm which is an iterative approach for approximating a convex function of one variable by piecewise linear functions [[RR92], [BHR91]]. It starts by evaluating the function and its one-sided derivatives at the endpoints of the given interval. The line connecting the two endpoints of the graph of the function yields an initial upper bound of the function, and the two supporting lines described by the derivatives at the end points give an initial lower bound of the function. Now, the procedure selects a point in the interval and evaluates the function and its derivative. (If the function is not differentiable, then any sub-gradient is taken). In this way, a better upper and lower approximation is achieved and the problem is split into two sub-intervals. Now, the sub-interval which has the larger error is selected and is partitioned in the same way as above. The process is continued for a given number of iterations or until a specified error bound is met. In Figure 3.3, the thin lines show the supporting lines described by the derivatives at each point in the function. The lower approximation of the function after two partitioning step is shown with the thick lines. In Figure 3.4, the thin lines shows the upper approximation after the first and second iterations. The lower approximation after the second partition in the third iteration is shown by the thick lines.

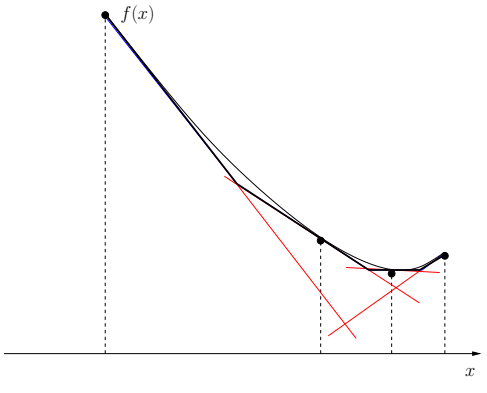


Figure 3.3: Lower approximation with the Sandwich algorithm after two partitioning steps shown by the thick lines.

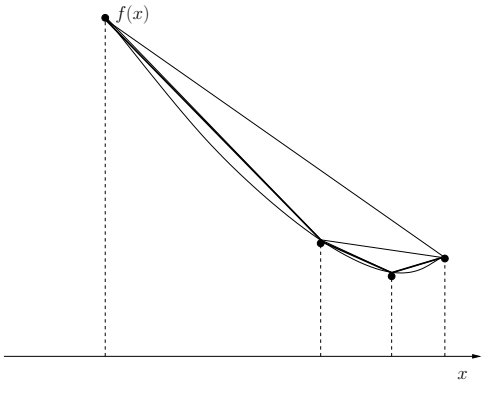


Figure 3.4: Upper approximation with the Sandwich algorithm after two partitioning steps. The thin lines show the approximation initially, after the first and after the second iteration respectively. The thick lines show the upper approximation of the function after three iterations.

There are different ways of how the interval is partitioned or in other words how to select a new point in the interval. [RR92] mentions the following four intuitive rules to choose

the new point :

- *The interval bisection rule*: The interval is partitioned into two equal parts.
- *The slope bisection rule*: We find the supporting line whose slope is the mean value of the slopes of the tangents at the endpoints. We partition the interval at some point where this line touches the function.
- *The maximum error rule*: The interval is partitioned at the breakpoint of the lower approximation, i.e., at the point where the error between the two approximations is maximum.
- *The chord rule*: We find the slope of the line connecting the endpoints of the interval. We partition the interval at some point where this line touches the function.

[RR92], [BHR91] proposed their algorithm with the purpose of approximating convex functions and shows the fast convergence of their approach. Our purpose is to converge to the minima of the convex function with as few function evaluations as possible. Although the end purpose is different our algorithm has the following similarities to the sandwich algorithm:

- Given a bracketing interval, we select a new sampling point to partition the interval into sub-intervals so that we find a new smaller bracketing interval. The sub-interval which has the larger error is selected for sampling.
- We find the lower approximation of the convex function with the function samples at the pivot points. Notice that we do not compute the derivatives of the function at the pivot points. For e.g., given 4 pivot points p_1, p_2, p_3 and p_4, we find the lower approximation of the function in the interval $[p_2, p_3]$ by extending the chords formed by connecting $f(p_1)$, $f(p_2)$ and $f(p_3)$, $f(p_4)$ respectively (Figure 3.5).

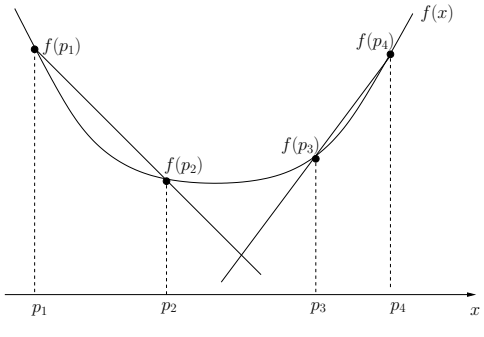


Figure 3.5: The extended chord connecting $f(p_1)$, $f(p_2)$ and $f(p_3)$, $f(p_4)$ gives a lower approximation of $f(x)$ in the interval $[p_2, p_3]$.

Our algorithm maintains a bracketing interval and an optimality gap at every iteration and this interval size and optimality gap decreases at each iteration, i.e., we converge towards the minima. The algorithm continues its iteration until either the minima is reached or the optimality gap is less than an acceptable error value. We shall explain the algorithm with the aid of a state machine. For better readability, we have divided

the entire state machine into smaller ones. Given pivots p_1, p_2, p_3 and p_4, we first compare $f(p_2)$ and $f(p_3)$ and based on the three possibilities, we know the exact bracketing interval. If $f(p_2) > f(p_3)$ then we are assured that the bracketing interval is $[p_2, p_4]$ and the algorithm is said to be in state $S1$. If $f(p_2) = f(p_3)$ then we know that the bracketing interval is $[p_2, p_3]$ and we denote this as state $S3$ in the algorithm, and finally if $f(p_2) < f(p_3)$ then the bracketing interval is $[p_1, p_3]$ and the algorithm is said to be in state $S2$ (Figure 3.6). We select a new sampling point in the bracketing interval and rename the pivots. In the following subsections, we describe each case in detail:

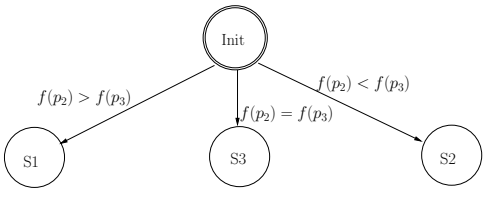


Figure 3.6: Sandwich algorithm as a state machine. Initial state to three possible states.

3.2.2.1 Sandwich Algorithm Illustration-1

We consider here the case when $f(p_2) > f(p_3)$. We find the lower approximation of $f(x)$ in the function domain interval $[p_2, p_3]$ by extending the chords formed by connecting the points $f(p_1)$, $f(p_2)$ and $f(p_3)$, $f(p_4)$. The point of intersection of the chords is denoted as say min_1. Similarly, we find the lower approximation of $f(x)$ in the domain interval $[p_3, p_4]$ by extending the chord formed by connecting the points $f(p_2)$, $f(p_3)$ and the vertical line passing through $f(p_4)$ and $(p_4,0)$. The point of intersection of the chord and the line is denoted as say min_2 (Figure 3.7).

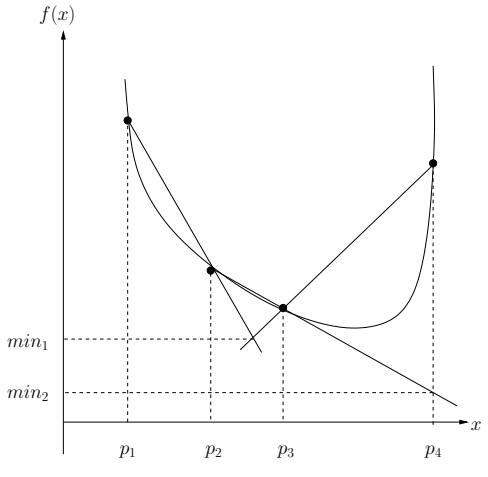


Figure 3.7: Lower approximation of $f(x)$ in the domain $[p_2, p_4]$ with extended chords.

Lower and Upper Bound We know that min_1 is the lower bound on $f(x)$ in the domain interval $[p_2, p_3]$ and similarly, min_2 is the lower bound on $f(x)$ in the domain interval $[p_3, p_4]$. Since we have $[p_2, p_4]$ as the bracketing interval in this case, we have $\min(min_1, min_2)$ as the lower bound on the minima of $f(x)$. The upper bound on the minima of $f(x)$ is min $(f(p_1), f(p_2), f(p_3)$ and $f(p_4))$ which in this case is $f(p_3)$.

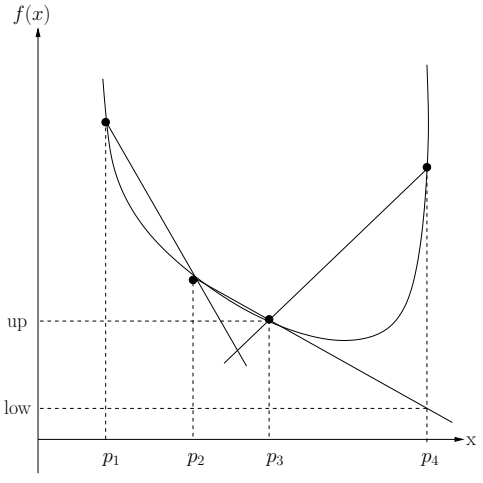


Figure 3.8: low and up denoting the lower and upper bound on the function minima.

Choice of Sub-interval Now that we have the lower and upper bound on the function minima, we choose the next sampling point. Out of the two intervals [p_2, p_3] and [p_3, p_4], we choose the interval for which the lower bound on the function is minimum. In Figure 3.7, min_1 is the lower bound of $f(x)$ in the domain interval [p_2, p_3], min_2 is the lower bound of $f(x)$ in the domain interval [p_3, p_4] and since $min_2 < min_1$, we choose the interval $[p_3, p_4]$ as the candidate to find a new sampling point. In the state machine, $min_1 \leq \min_2$ depicts state S11 and $min_2 < min_1$ is depicted by state S12 which is shown in Figure 3.10. This choice is natural in the sense that since the lower bound is lower, there is more chance that the function minima will be contained in the interval (Greedy choice). if both min_1 and min_2 are the same then we choose any of the interval arbitrarily.

Point Selection Rule After the interval is chosen, we have different possibilities of how to choose a new sampling point in the interval. We use **bisection rule** here (Figure 3.9). Observe that the maximum error rule is not a good choice here because it will lead to p_4 as the new evaluation point which we already have. We do use maximum error rule for choosing the new point in conditions described later in a subsection.

We also experimented by choosing the new sampling point at not the midpoint but at the golden section [at (1 - *golden ratio*) of the interval] to observe if the asthetics of this number makes the algorithm converge faster to the minima. We did not notice any mentionable difference.

Pivots Renaming Depending on the function value at this new point, we rename the pivots and start a new iteration. We consider both possibilities of having [p_2, p_3] and [p_3, p_4] as the selected interval of sampling. Let us say that p is the new point. First consider that [p_2, p_3] is the chosen sub-interval, i.e., algorithm is in state S11. if $f(p) < f(p_3)$ then we know that [p_2, p_3] is the bracketing interval and [p_1, p_2] and [p_3, p_4] can be discarded. We keep all the 5 pivots with renaming and move to 5 pivots state instead which is depicted by the state S4 in the state machine. p is renamed to p_3, p_3 is renamed to p_4 and p_4 is renamed to p_5. if $f(p) = f(p_3)$ then we know that [p, p_3] is the bracketing interval. p_2 is renamed to p_1, p to p_2 and the algorithm moves to state S3. Finally, if $f(p) > f(p_3)$, p_2 is renamed to p_1, p is renamed to p_2 and the algorithm moves to the state S1 again.

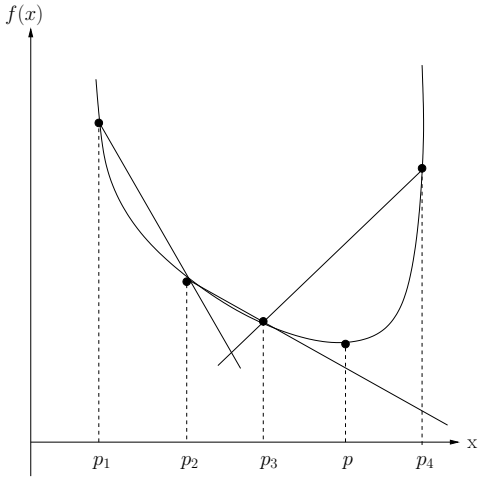


Figure 3.9: New pivot point p selected by bisecting interval $[p_3, p_4]$

Now consider that $[p_3, p_4]$ is the interval of choice, i.e., the algorithm is in the state S12. If $f(p) < f(p_3)$, p_2 is renamed to p_1, p_3 to p_2 and p to p_3 (Figure 3.12) and the algorithm moves to the state S1. if $f(p) = f(p_3)$ then we know that $[p_3, p]$ is the bracketing interval. p_2 is renamed to p_1, p_3 is renamed to p_2 and p is renamed to p_3. The algorithm moves to the state S3. If $f(p) > f(p_3)$ then we know that $[p_2, p]$ is the bracketing interval and $[p_1, p_2]$ and $[p_3, p_4]$ can be discarded. We keep all the five pivots with renaming and move to five pivots state. p is renamed to p_4 and p_4 is renamed to p_5 and the algorithm moves to the state S4. Figure 3.10 shows the state machine starting from the state S1.

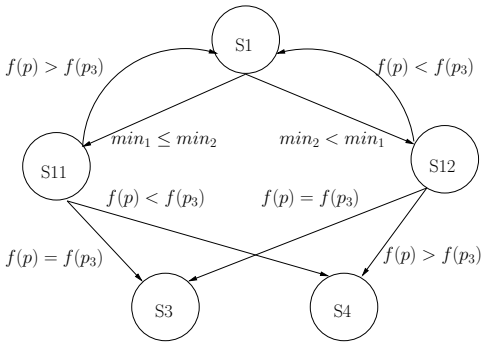


Figure 3.10: State machine with state S1 as the starting state.

3.2.2.2 Sandwich Algorithm Illustration-2

Let us consider the case when $f(p_2) = f(p_3)$. In this situation, $[p_2, p_3]$ is the bracketing interval. We find the lower approximation of $f(x)$ in the function domain interval $[p_2, p_3]$ by extending the chords formed by connecting the points $f(p_1)$, $f(p_2)$ and $f(p_3)$, $f(p_4)$. Let p denote the abscissa and low denote the ordinate of the point of intersection of the two chords.

Lower and Upper Bound Since $[p_2, p_3]$ is the bracketing interval and we have low as the lower bound of the function in the domain interval $[p_2, p_3]$. Hence low is the lower bound on the function minima. The upper bound on the minima of $f(x)$ is min ($f(p_1)$, $f(p_2)$, $f(p_3)$ and $f(p_4)$) (Figure 3.13).

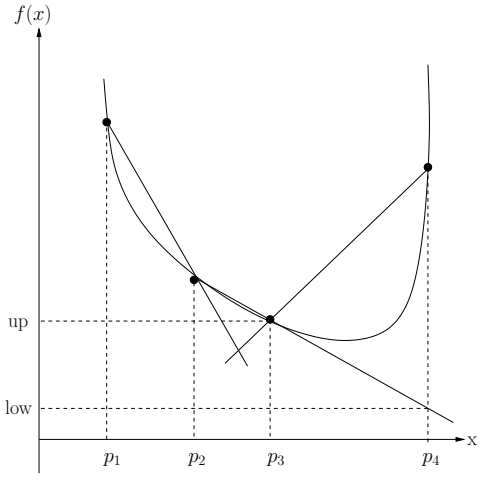


Figure 3.11: low and up denoting the lower and upper bound on the function minima.

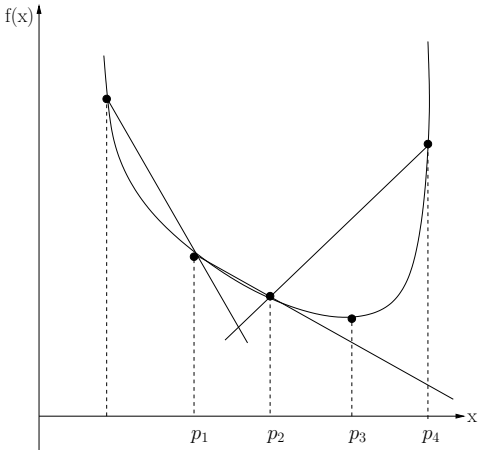


Figure 3.12: After renaming the pivots.

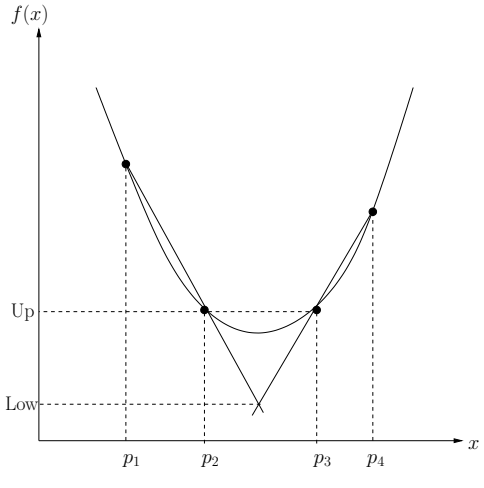


Figure 3.13: Lower and Upper bound on the function minima.

Point Selection Rule Since we already know the bracketing interval in this case, we simply need to choose a new sampling point in this interval. We use the **maximum error rule** here as the new sampling point selection rule. In Figure 3.14, point p denotes

the new sampling point.

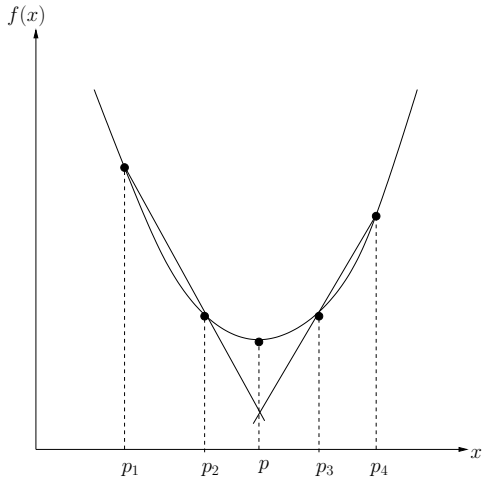


Figure 3.14: Selecting a new point for evaluation with maximum error rule.

Pivots Renaming If the function value at this new point, i.e., $f(p)$ is the same as $f(p_2)$ or $f(p_3)$ then using corollary 3.1 we find the function minima $f_{\min} = f(p) = f(p_1) = f(p_2)$ and the algorithm terminates and moves to the Stop state. Otherwise, $f(p)$ has to be less than $f(p_2)$ or $f(p_3)$ since $f(2)$, $f(3)$ gives the upper bound on $f(x)$ in the domain interval $[p_2, p_3]$. In this case, we rename p to p_3, p_3 to p_4 and p_4 to p_5 and move to the five pivots state (Figure 3.15). Figure 3.16 shows the state machine starting from the state S3.

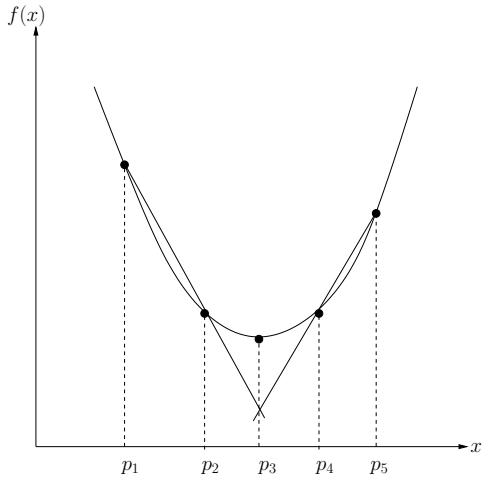


Figure 3.15: Renaming the pivots.

3.2.2.3 Sandwich Algorithm Illustration-3

Let us consider the case when $f(p_2) < f(p_3)$. This is only a symmetric situation of $f(p_2) > f(p_3)$. Figure 3.17 shows the state machine starting from the state S2.

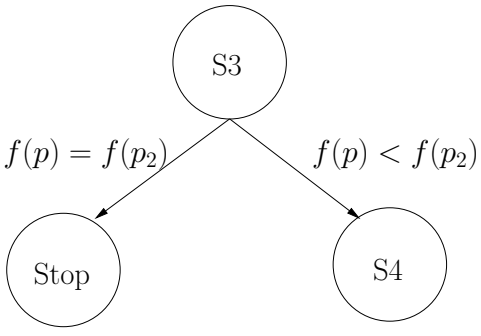


Figure 3.16: State machine with state S3 as the starting state.

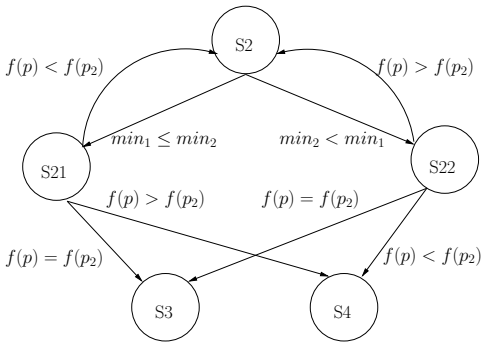


Figure 3.17: State machine with state S2 as the starting state.

3.2.2.4 Sandwich Algorithm Illustration-4

We call this case the five pivots state. Five pivots state is shown in Figure 3.18. In this situation, we know that $[p_2, p_4]$ is the bracketing interval. We keep five pivots because using them, we can hope to find a tighter lower approximation of $f(x)$ in the domain interval $[p_2, p_4]$. The chord connecting the points $f(p_1)$, $f(p_2)$ when extended beyond p_2 gives a lower approximation on $f(x)$ and similarly, the chord connecting the points $f(p_4)$, $f(p_3)$ when extended beyond p_3. The point of intersection of these two extended chords gives a lower bound on the minima of $f(x)$ in the domain interval $[p_2, p_3]$. let min_1 denote the ordinate of this intersection point and x_1 denote the abscissa. In the same way, the chord connecting the points $f(p_2)$, $f(p_3)$ and the chord connecting the points $f(p_5)$, $f(p_4)$ gives a lower approximation of $f(x)$ in the domain interval $[p_3, p_4]$ and the point of intersection of these two extended chords gives a lower bound on the function minima in this domain. let min_2 denote the ordinate and x_2 the abscissa of the intersection point.

Lower and Upper Bound Since $[p_2, p_4]$ is the bracketing interval and we have min_1 to be the lower bound on the minima of $f(x)$ in the domain interval $[p_2, p_3]$, min_2 to be the lower bo'und in the domain interval $[p_3, p_4]$ the lower bound on the minima of $f(x)$ is $\min(min_1, min_2)$. The upper bound on the minima of $f(x)$ is $f(p_3)$. For example, in Figure 3.18, min_2 gives the lower bound on the minima of $f(x)$.

Proposition 3.1. *If $min_1 = min_2 = f(p_3)$ then $f_{\min} = f(p_3)$, where $f_{\min}$ denotes the minima of function $f(x)$.*

Proof. Let $min_1 = min_2 = f(p_3)$. Since min_1 is the lower bound on $f(x)$ in $[p_2, p_3]$ and

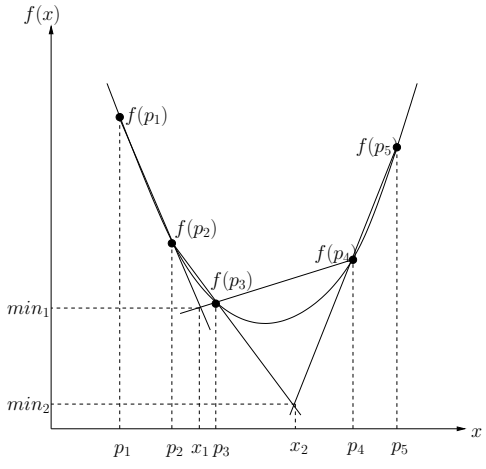


Figure 3.18: Lower approximation of $f(x)$ in $[p_2, p_4]$ by extending chords $(f(p_1), f(p_2))$, $(f(p_4), f(p_3))$ and $(f(p_2), f(p_3))$, $(f(p_5), f(p_4))$

min_2 is the lower bound on $f(x)$ in $[p_3, p_4]$ and as $min_1 = min_2$, we have $min_1 = min_2$ as the lower bound on $f(x)$ in $[p_2, p_4]$. Also, as $[p_2, p_4]$ is the bracketing interval, we have $min_1 = min_2$ as the lower bound on the function minima $f_{\min}$. As we have $f(p_3) = min_1 = min_2$, it must be the case that $f(p_3) = f_{\min}$. □

We check the above condition to terminate the iteration process and return the function minima.

Choice of Sub-interval As before, we make a greedy choice. Out of the two intervals $[p_2, p_3]$ and $[p_3, p_4]$, we choose the interval for which the lower bound on the function is minimum . If $[p_2, p_3]$ is the chosen interval of sampling, the algorithm goes to state S41 and if $[p_3, p_4]$ is the chosen interval, the algorithm goes to state S42. The algorithm terminates if $min_1 = min_2$ using the proposition 3.1 (Figure 3.19). For e.g., in Figure 3.18, since $min_2 < min_1$, we choose the interval $[p_3, p_4]$ as the candidate to find a new sampling point. If the lower bound is the same in both the intervals, i.e., $min_1 = min_2$ (but not equal to $f(p_3)$, otherwise $f_{\min} = f(p_3)$) then the one of the two intervals is chosen arbitrarily for sampling.

Point Selection Rule In this case also, we use the **maximum error rule** for the new sampling point selection. In Figure 3.18, point x_2 denotes the new sampling point.

Pivots Renaming We consider both the possibilities of the chosen sub-interval for finding the new sampling point and describe the pivot renaming. Lets consider that $[p_2, p_3]$ is the chosen interval for finding the new sampling point and x_1 as the new sampling point. If $f(x_1) > f(p_3)$, we remain in the five pivots state by renaming the pivots p_2 to p_1 and x_1 to p_2. If $f(x_1) = f(p_3)$, we know that the bracketing interval is $[x_1, p_3]$. We discard the intervals $[p_1, p_2]$ and $[p_4, p_5]$. p_2 is renamed to p_1 and x_1 is renamed to p_2. Finally, if $f(x_1) < f(p_3)$, we discard the interval $[p_4, p_5]$. The pivots are renamed as x_1 to p_3, p_3 to p_4 and p_4 to p_5 and we again have another five pivots state.

Now, lets consider the other possibility of $[p_3, p_4]$ being the chosen sub-interval and x_2 as the new sampling point. If $f(x_2) > f(p_3)$, we discard the interval $[p_3, p_4]$ for further

search and the pivots are renamed as x_2 to p_4 and p_4 to p_5. We have a five pivots state again. If $f(x_2) = f(p_3)$, we know that the bracketing interval is $[p_3, x_2]$. We discard the intervals $[p_1, p_2]$ and $[p_4, p_5]$ for further search. p_2 is renamed to p_1, p_3 to p_2 and x_2 is renamed to p_3. Finally, if $f(x_2) < f(p_3)$, we discard the interval $[p_1, p_2]$. The pivots are renamed as p_2 to p_1, p_3 to p_2, x_2 to p_3 and we have a five pivots state. Figure 3.19 shows the state machine starting from the state S4.

There is a situation for which we find the function minimum and we terminate the algorithm. This is under the condition that $min_1 = min_2 = min$ (say) and $f(x_1)$ or $f(x_2)$ equals min. In this case, min is the minima because it is the lower bound on $f(x)$ in the domain interval $[p_2, p_4]$ and $\exists x \in [p_2, p_4]$ such that $f(x) = min$. Hence, min is indeed the function minima.

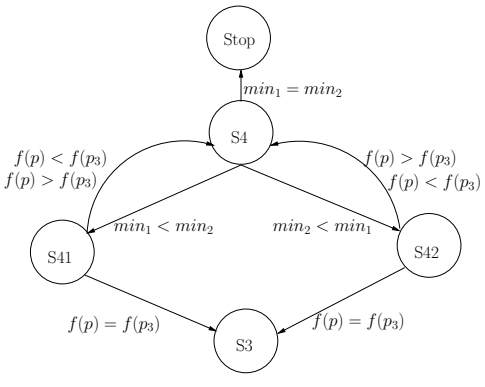


Figure 3.19: State machine with state S4 as the starting state.

The *sandwich algorithm* is illustrated in algorithm 3.4. We combine our *minima bracketing* and our *sandwich algorithm* to have a novel minimization algorithm for convex functions. We name our minimization algorithm as **Lower Bound Search** algorithm. The name derives from the fact that our minima search is based in some sense on the comparisons of the lower bounds of the function at different domain intervals. We explained above these lower bounds on the functions by means of extended chords with end points of the chord on the function. We now try to put it more formally.

We compute a lower bound function $f^-(\lambda) \leq f(\lambda)$ [Figure 3.20], which we update with each newly computed sample in our algorithm. Given two samples $(\lambda_i, f(\lambda_i))$ and $(\lambda_j, f(\lambda_j))$, $\lambda_i < \lambda_j$, the convexity of $f(\lambda)$ implies that the straight line through them,

$$f_{ij}^-(\lambda) = \frac{f(\lambda_j) - f(\lambda_i)}{\lambda_j - \lambda_i}(\lambda - \lambda_i) + f(\lambda_i), \tag{3.15}$$

is a lower bound on $f(\lambda)$ to the left and right of the two points, i.e., for all $\lambda \leq \lambda_i$ and $\lambda \geq \lambda_j$, and an upper bound between them, i.e., for $\lambda_i \leq \lambda \leq \lambda_j$. We combine (3.15) for all known samples $(\lambda_i, f(\lambda_i))$ to the following lower bound function, which is defined pointwise over λ:

$$f^-(\lambda) = \max\Big(-\infty, \max_{\lambda \leq \lambda_i < \lambda_j} f_{ij}^-(\lambda), \max_{\lambda_i < \lambda_j \leq \lambda} f_{ij}^-(\lambda)\Big). \tag{3.16}$$

We compute an interval $[r_-, r_+]$ containing $\min_{\lambda \in \mathbb{R}} f(\lambda)$, whose *optimality gap* $r_+ - r_-$ is smaller than a given threshold $\varepsilon \geq 0$. *Lower Bound search* proceeds as follows:

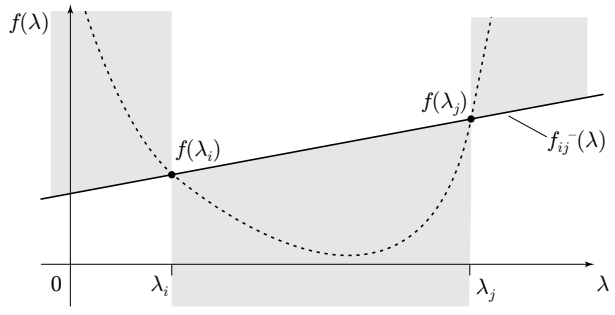


Figure 3.20: The straight line through two points on a convex function $f(\lambda)$ is a lower bound on $f(\lambda)$ to the left and to the right of those two points.

1. Let $i = 0$, $\lambda_i = 0$, $\lambda_{i+1} = 1$, $r_- = -\infty$, $r_+ = +\infty$.
2. Bracket the minimum by adding samples until a turning point is found, i.e., $f(\lambda_{i-1}) \leq f(\lambda_{i-2})$ and $f(\lambda_{i-1}) \leq f(\lambda_i)$, increasing the distance between λ_i exponentially.
3. Compute $f(\lambda_i)$ and tighten the interval bounds $r_- \leftarrow \inf_{\lambda \in \mathbb{R}^{\geq 0}} f^-(\lambda)$, $r_+ \leftarrow \min(r_+, f(\lambda_i))$
4. Choose the next sample at the lowest point of $f^-(\lambda)$ unless already visited:
   (a) Let $\lambda_{i+1} \leftarrow \operatorname{arginf}_{\lambda \in \mathbb{R}^{\geq 0}}(f^-(\lambda))$.
   (b) If $\lambda_{i+1} \in \{\lambda_0, \ldots, \lambda_i\}$, let $\lambda_{i+1} \leftarrow (\lambda_{i+1} + \lambda_j)/2$, where λ_j is the appropriate neighboring sample.
5. If $r_+ - r_- > \varepsilon$, let $i \leftarrow i + 1$ and go to (3).

Algorithm 3.4 Sandwich Algorithm

Require: $p_1 < p_2 < p_3 < p_4$ and $f(p_1) > f(p_2)$ and $f(p_3) < f(p_4)$, $\varepsilon \in \mathbb{R}$
Ensure: $[l, u]$ such that $l < f_{\min} < u$ and $u - l <= \varepsilon$
1: $p_5 = p_4$ {fifth pivot is set to p_4 until the five pivots state is reached}
2: *update_bounds*$(p_1, p_2, p_3, p_4, p_5)$.
3: **while** $u - l > \varepsilon$ **do**
4: $[x, y] \leftarrow$ *choose_sampling_interval*$(p1, p2, p3, p4, p5)$.
5: $s \leftarrow$ *sample_interval*$([x, y])$.
6: *rename_pivots*(s).
7: *update_bounds*$(p_1, p_2, p_3, p_4, p_5)$.
8: **end while**

Chapter 4

Flowpipe-Guard Intersection with Support Functions

Given an outgoing transition from the source location of the automaton, we collect all the guard constraints, source invariant constraints and the constraints of the pre-map of the target invariant in say, $\mathcal{G}$. The reason for including the source and the pre-mapped target invariant constraints is for higher precision explained in section 2.4.2. We assume that the constraints are all linear constraints, equality or inequality. The conjunction of the constraints in $\mathcal{G}$ defines an H-polyhedron. The Ω_i of the flowpipe defined in (2.28) are support function represented convex sets. Note that not all the N Ω_i of the flowpipe necessarily intersect with the guard set $\mathcal{G}$. Therefore, the detection of intersection is foremost. We then compute the intersection of the guard set with the section(s) of the flowpipe that intersect with it.

Let us mention that we simplify the problem of intersecting a flowpipe with a polyhedral guard set by considering the following sub-problems:

- Intersecting a convex set with a hyperplane/halfspace.
- Intersecting a collection of convex sets with a hyperplane/halfspace.
- Intersecting a collection of convex sets with a polyhedron.

The intersection sets are represented by their support functions. We make polyhedral approximations from the support function representation whenever the need be by sampling the support function in the desired directions. In the next section we illustrate how the intersection of a flowpipe with a guard set is detected and an illustration is presented with an example. The subsequent sections describe the support function representation of the intersection of a convex set with a hyperplane or halfspace, collection of convex sets with a hyperplane or halfspace and the intersection of a collection of convex sets with a polyhedron respectively.

4.1 Detecting Intersection of a Guard with a Flowpipe

We detect the intersection of Ω_i with the guard set $\mathcal{G}$. For that, we identify indices $i \leq N$ of the flowpipe such that $\Omega_i \cap \mathcal{G} \neq \emptyset$. We define a list of intervals $\mathcal{I}$ having intervals $[i_{\min}, i_{\max}]$ of the indices of the flowpipe such that $\Omega_i \cap \mathcal{G} \neq \emptyset$, $\forall i \in [i_{\min}, i_{\max}]$. The computation of $\mathcal{I}$ proceeds as follows:

1. $\mathcal{I} = [i_0, i_N]$
2. $\forall g \in \mathcal{G}$, $\mathcal{I}'$ = List of intervals $[i_{\min}, i_{\max}]$ such that $\Omega_i \cap g \neq \emptyset, \forall i \in [i_{\min}, i_{max}]$.
3. $\mathcal{I} = \mathcal{I} \cap \mathcal{I}'$ and go to (2).

Where intersection of list of intervals is defined as pairwise intersection.

For hyperplanar constraint $\mathcal{H} = \{x : x.n = \lambda\}$, we use the method described in [LG09] to detect intersection. We use a similar approach for detecting intersection with halfspace constraints. For hyperplanar constraints, we use the following lemma in the computation of $\mathcal{I}$:

Lemma 4.1. *Given a hyperplane $\mathcal{H} = \{x \in \mathbb{R} : x.n = \gamma\}$ and a compact convex set $\mathcal{X}$, we have*

$$\mathcal{X} \cap \mathcal{H} \neq \emptyset \iff -sup_{\mathcal{X}}(-n) \leq \gamma \leq sup_{\mathcal{X}}(n). \tag{4.1}$$

Proof. We first prove that:

$$-sup_{\mathcal{X}}(-n) \leq \gamma \leq sup_{\mathcal{X}}(n) \implies \mathcal{X} \cap \mathcal{H} \neq \emptyset.$$

Let $-sup_{\mathcal{X}}(-n) \leq \gamma \leq sup_{\mathcal{X}}(n)$.
$\implies \min_{x \in \mathcal{X}} (n.x) \leq \gamma \leq \max_{x \in \mathcal{X}} (n.x)$.

Consider the case when $\min_{x \in \mathcal{X}} (n.x) = \gamma$ or $\max_{x \in \mathcal{X}} (n.x) = \gamma$
$\implies \exists x \in \mathcal{X}$ such that $n.x = \gamma$.
$\implies \exists x \in \mathcal{X}$ such that $x \in H$.
$\implies \exists x \in \mathcal{X} \cap \mathcal{H}$.
$\implies \mathcal{X} \cap \mathcal{H} \neq \emptyset$.

Now, consider that $\min_{x \in \mathcal{X}} (n.x) < \gamma < \max_{x \in \mathcal{X}} (n.x)$.
Let $x_1 \in \mathcal{X}$ such that $\min_{x \in \mathcal{X}} (n.x) = x_1.n < \gamma$
and let $x_2 \in \mathcal{X}$ such that $\max_{x \in \mathcal{X}} (n.x) = x_2.n > \gamma$.

Let $x_1.n = \gamma + k_1$ and $x_2.n = \gamma - k_2$, k_1, $k_2 \in \mathbb{R}^+$.
Since $\mathcal{X}$ is a convex set, by convexity property, we know that:

$$y = \lambda x_1 + (1 - \lambda x_2) \in \mathcal{X}, \forall \lambda \in [0, 1].$$

It can be shown that for $\lambda = k_2/(k_1 + k_2)$, $y.n = \gamma$.

Therefore, $\exists y \in \mathcal{X}$ such that $y \in \mathcal{H} \implies \mathcal{X} \cap \mathcal{H} \neq \emptyset$.

Now, we prove that:

$$\mathcal{X} \cap \mathcal{H} \neq \emptyset \implies -sup_{\mathcal{X}}(-n) \leq \gamma \leq sup_{\mathcal{X}}(n)$$

Let $\mathcal{X} \cap \mathcal{H} \neq \emptyset$. Then we can partition the set $\mathcal{X}$ into three sets as:

$\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3$ such that:
$\mathcal{X}_1 = \{x \in X | n.x < \gamma\}$.
$\mathcal{X}_2 = \{x \in X | n.x = \gamma\}$.
$\mathcal{X}_3 = \{x \in X | n.x > \gamma\}$.

Since $\mathcal{X} \cap \mathcal{H} \neq \emptyset$, $\mathcal{X}_2 \neq \emptyset$.
We can see that $\max_{x \in \mathcal{X}} (n.x) \geq \gamma$, equality holds when $\mathcal{X}_3 = \emptyset$ and $\min_{x \in \mathcal{X}} (n.x) \leq \gamma$, equality holds when $\mathcal{X}_1 = \emptyset$.

Therefore, $\min_{x \in \mathcal{X}} (n.x) \leq \gamma \leq \max_{x \in \mathcal{X}} (n.x)$.
$\implies -sup_{\mathcal{X}}(-n) \leq \gamma \leq sup_{\mathcal{X}}(n)$.

We prove both directions of the $\iff$ condition and hence the result is proved.

□

For halfspace constraints, we use the following lemma in the computation of $\mathcal{I}$.

Lemma 4.2. *Given a Halfspace $\mathcal{H} = \{x \in \mathbb{R} : x.n \leq \gamma\}$ and a compact convex set $\mathcal{X}$, we have*

$$\mathcal{X} \cap \mathcal{H} = \emptyset \iff -sup_{\mathcal{X}}(-n) > \gamma. \tag{4.2}$$

Proof. Let $\mathcal{X} \cap \mathcal{H} = \emptyset$.

$$\begin{aligned} \mathcal{X} \cap \mathcal{H} = \emptyset &\implies \forall x \in \mathcal{X}, n.x > \gamma \\ &\implies \forall x \in \mathcal{X}, -n.x < -\gamma \\ &\implies \max_{x \in \mathcal{X}} (-n.x) < -\gamma \\ &\implies sup_{\mathcal{X}}(-n) < -\gamma \\ &\implies -sup_{\mathcal{X}}(-n) > \gamma \end{aligned} \tag{4.3}$$

Let $-sup_{\mathcal{X}}(-n) > \gamma$.

$$\begin{aligned} -sup_{\mathcal{X}}(-n) > \gamma &\implies \min_{x \in \mathcal{X}} (n.x) > \gamma \\ &\implies \forall x \in \mathcal{X}, n.x > \gamma \\ &\implies \mathcal{X} \cap \mathcal{H} = \emptyset \end{aligned} \tag{4.4}$$

(4.3) and (4.4) proves the lemma. □

Using the above lemmas to check for the emptiness, we can identify the set of intersecting intervals $\mathcal{I}$.

Example 4.1. *Figure 4.1(a) illustrates the flowpipe approximation of a five dimensional system. The boxes represent the polyhedral approximation of the support function represented flowpipe in the axes directions. Each box in the figure represents a member Ω of the flowpipe. Figure 4.1(b) shows the sections of the flowpipe which intersects with the guard $y = 0$. Figure 4.1(c) shows the sections of the flowpipe intersecting with the halfspace $y \leq 0$.*

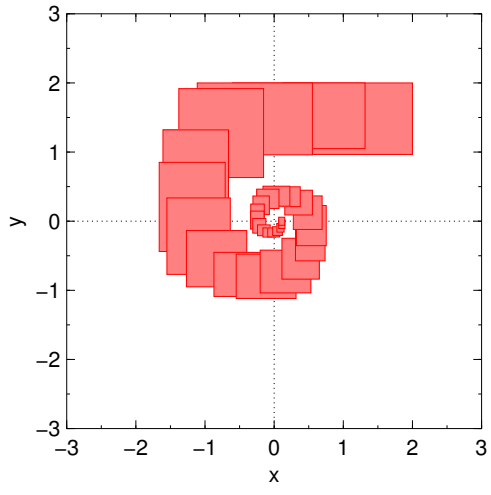
Algorithm 4.1 pseudo code of discrete_post

Require: ha_aut, loc

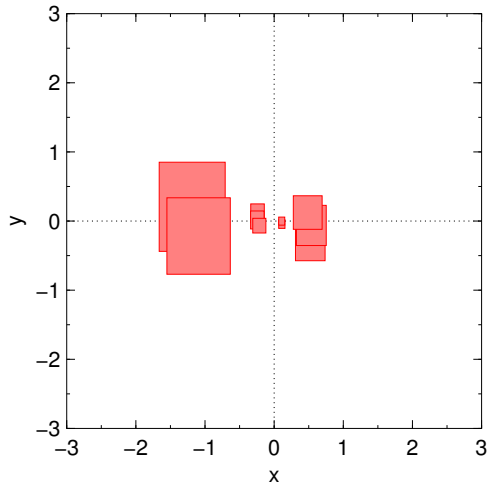
```
1: trans ← ha_aut.get_out_transitions(loc)
2: flowpipe ← loc.continuous_post();
3: while t ∈ trans do
4:     tgt ← t.get_target_loc();
5:     cont_set ← t.guard ∩ flowpipe {cont_set is a continuous set}
6:     cont_set ← t.assign_map(cont_set)
7:     cont_set ← tgt.get_invariant() ∩ cont_set;
8:     sym_state ← symbolic_state(tgt, cont_set)
9:     PLWL.add(sym_state) {sym_state is a symbolic state which is pair of location and
       continuous set. PLWL is the passed and waiting list}
10: end while
```

4.2 Intersecting a Convex Set with a Hyperplane or Halfspace

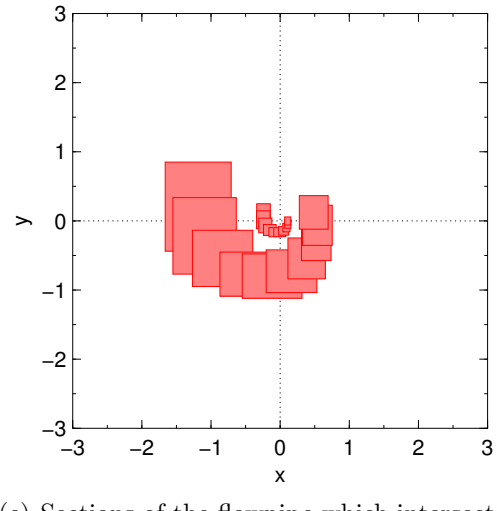
To have a support function representation of the intersection between a convex set and a hyperplane or halfspace, we must know how to compute its support function. Section 3.1 shows that the support function computation of the intersection between a convex set and a hyperplane or halfspace reduces to the problem of minimizing a convex function. We presented our novel approach to the minimization problem in previous chapter 3 which we use to have the support function representation of the intersection set. Before we apply the minimization algorithm, we can get rid of the term $\lambda\gamma$ from $f(\lambda)$ in (3.5), for computational simplicity, by shifting of the operand sets as discussed below.



(a) Flowpipe approximation of a five dimensional system.



(b) Sections of the flowpipe which intersects with the hyperplanar guard $y = 0$.



(c) Sections of the flowpipe which intersects with the halfspace guard $y \leq 0$.

Figure 4.1: Flowpipe sections intersecting with a hyperplane and a halfspace illustrating the intersection detection algorithm.

4.2.1 Shifting the Convex Set and the Hyperplane or Halfspace

For computational simplification, we shift the hyperplane or halfspace making it pass through the center, making its distance from the center to be 0 and hence dismissing the term $\lambda\gamma$, γ being 0 in (3.5). We also apply the same shift to the convex set $\mathcal{S}$ to get the same intersection set, but shifted.

Given a guard $\mathcal{G} : x.n \bowtie \gamma$, $\bowtie \in \{<, =\}$, we compute the translation vector, say b, as :

$$b = [\gamma/norm(n)].(n/norm(n)); \tag{4.5}$$

After we have the translation vector b, We use (4.7) to compute the required support function which is derived from the property of support functions given in (4.6).

$$sup_{(\mathcal{S}\cap\mathcal{G})\oplus b}(l) = sup_{\mathcal{S}\cap\mathcal{G}}(l) + (b.l) \tag{4.6}$$
$$sup_{\mathcal{S}\cap\mathcal{G}}(l) = sup_{(\mathcal{S}\cap\mathcal{G})\oplus b} - (b.l) \tag{4.7}$$

After the shifting, $f(\lambda)$ of 3.5 reduces to $f(\lambda) = sup_{\mathcal{X}'}(\ell - \lambda n)$, which is the support function of a compact convex set $\mathcal{X}$ in the *lambda domain* as seen in section 2.2.3. We already mentioned above that $f(\lambda)$ is a convex piecewise linear function for polyhedral sets. The support function graphs of an hexagon and a polytope with 15 facets in section 2.2.3 gives the reader an idea of the nature of the function.

Lower Bound Search algorithm finds a sequence of sampling points λ_i that converges towards the minimum of $f(\lambda)$, see Figure 4.2 and Figure 4.3 for an illustration. Each λ_i corresponds to the normed direction

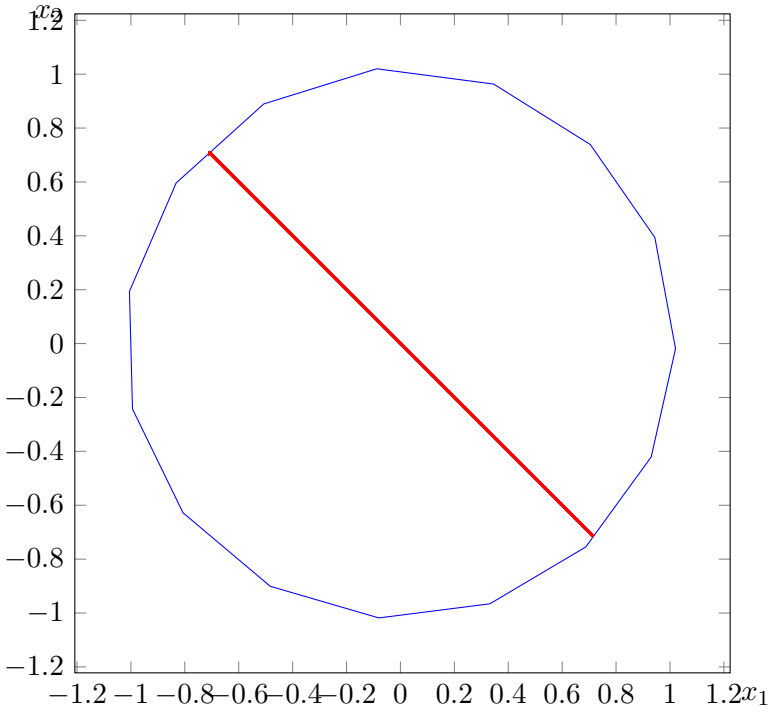
$$\hat{\ell}_i = c_i(\ell - \lambda_i n), \text{ with } c_i = 1/\|\ell - \lambda_i n\|_2.$$

4.2.2 Related Work

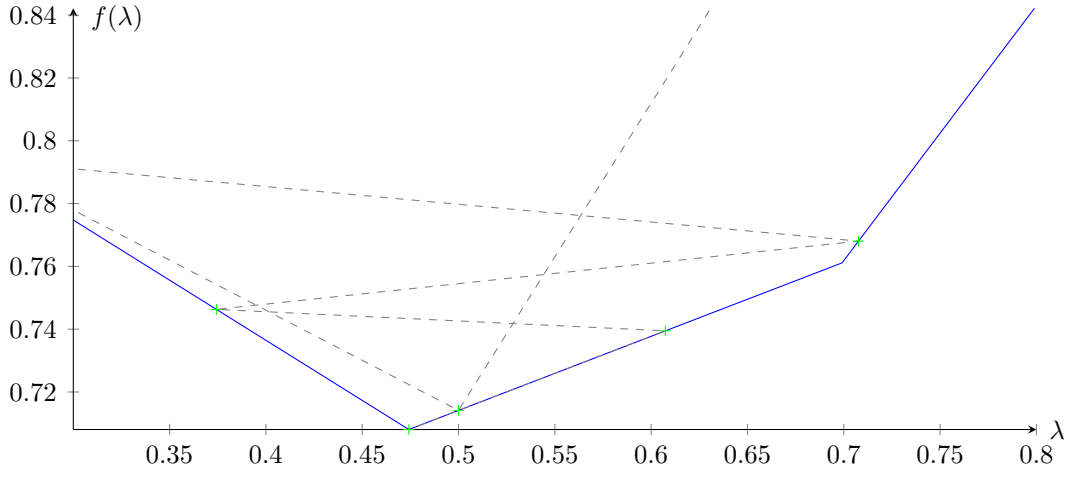
To the best of our knowledge, this is the first proposed solution for a support function representation of the intersection of a convex set with a halfspace or polyhedron. It is derived from previous work on the intersection with a hyperplane by [GG09]. There, the support function of the intersection is reduced to a univariate minimization problem that is derived geometrically. Its parameter $\theta \in (0, \pi)$ describes the angle between the sample direction and the normal vector of the hyperplane $\mathcal{H}' = \{x \mid nx = \gamma\}$:

$$sup_{\mathcal{X}\cap\mathcal{H}'}(\ell) = \inf_{\theta\in(0,\pi)} \frac{sup_{\mathcal{X}}\,(\ell \sin\theta + n\cos\theta) - \gamma\cos\theta}{\sin\theta}. \tag{4.8}$$

While (4.8) has the advantage over (3.6) that its argument ranges over a finite interval, its cost function is unimodal instead of convex. Therefore one has no direct estimate of the optimality gap, and it is not possible to obtain the exact solution. Recall that if $\mathcal{X}$ is a polytope, Lower Bound Search computes the exact solution of (3.6) in a finite number of steps. We refer to the approximate solution of (4.8) by golden section search as *Golden Section Search in the Polar Domain* (GSPD).



(a) The polytope $\mathcal{P}$ and its intersection with the hyperplane $\mathcal{H}'$



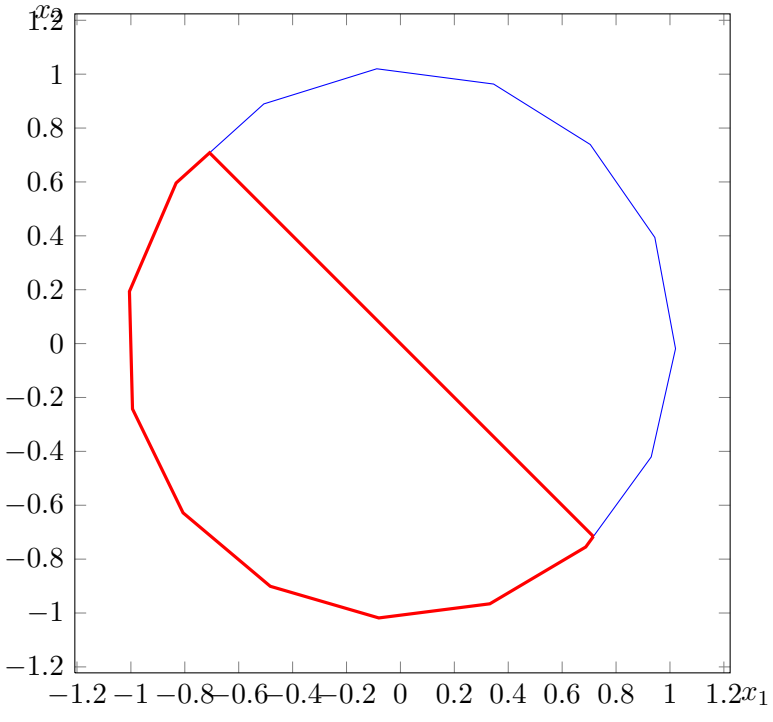
(b) To compute $sup_{\mathcal{X}\cap\mathcal{H}'}(\ell)$, we minimize $f(\lambda)$. The function and the samples chosen by the Lower Bound Search are shown for $\ell = (0, 1)$

Figure 4.2: Intersection of the hyperplane $\mathcal{H}' = \{x + y = 0\}$ with a polytope $\mathcal{P}$ with 15 facets

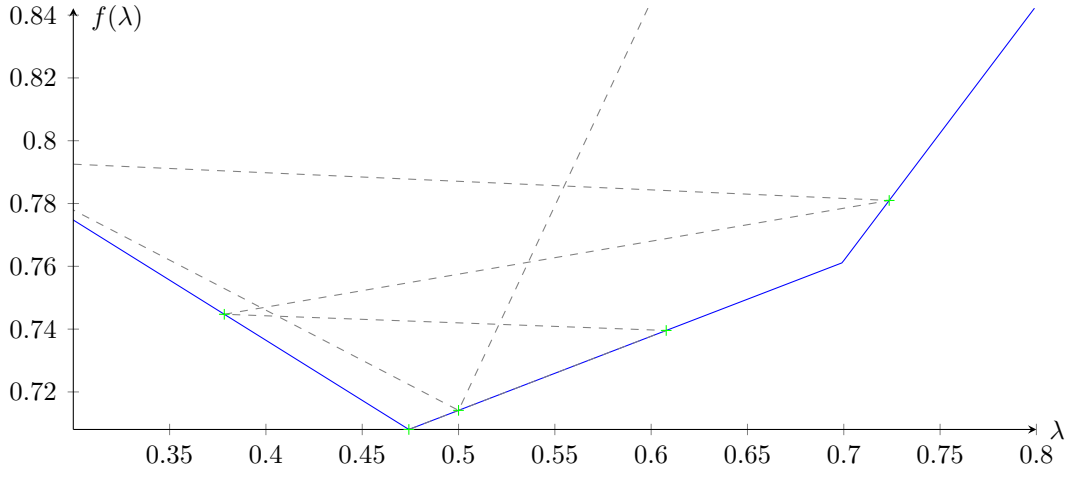
4.2.3 Experiments

The following experiments illustrate the performance of Lower Bound Search in comparison with GSPD.

Table 4.1 compares the support function computation of the intersection between a regular n-polyhedron in two dimensions with the line $x\cos\theta + y\sin\theta = 0$ in the direction $[0, 1]$, for 1000 uniformly distributed $\theta \in [0, \pi]$ by GSPD and Lower Bound Search. The table shows the averaged results. Note that the error of GSPD decreases as the number of facets increases. The reason is that the support function becomes flatter near



(a) The polytope $\mathcal{P}$ and its intersection with the halfspace $\mathcal{H}$



(b) To compute $sup_{\mathcal{X}\cap\mathcal{H}}(\ell)$, we minimize $f(\lambda)$. The function and the samples chosen by the Lower Bound Search are shown for $\ell = (0, 1)$

Figure 4.3: Intersection of the halfspace $\mathcal{H} = \{x + y \leq 0\}$ with a polytope $\mathcal{P}$ with 15 facets

the minimum for polyhedra with larger number of facets. Hence for a fixed interval in the function domain bracketing the minimum, the difference between the minimum and the upper bound decreases. Table 4.2 compares the intersection between a regular n-polyhedron with the line $x\cos\theta + y\sin\theta = 0$ in the direction $[0, 1]$, for 1000 uniformly distributed $\theta \in [0, \pi]$ for a fixed number of samples. The table shows the averaged results.

Remark 4.1. Note that in Table 4.2 the computation times differ even though the same number of samples is computed for both LBS and GSPD. Indeed the computation time of a sample is data as well as state dependent. In particular, the LP solver computing the support function keeps its state between calls. A sample can therefore be computed

Table 4.1: Average performance of Lower Bound Search (exact solution) vs GSPD (fixed to 14 samples), intersecting a hyperplane with a polytope

	Lower Bound			GSPD		
facets	samples	err	time(ms)	samples	err$\times 10^{-4}$	time(ms)
4	6.741	0	0.15	14	8.197	0.71
8	8.523	0	0.34	14	3.200	0.82
16	9.611	0	0.50	14	1.612	1.27
24	10.222	0	0.74	14	1.111	1.80

Table 4.2: Average performance of Lower Bound Search vs GSPD, intersecting a hyperplane with a polytope for a fixed number of samples (6)

	Lower Bound			GSPD	
Facets	opt. gap	err	time(ms)	err	time(ms)
4	0.0338614	0.0285933	0.16	0.0351516	0.25
8	0.0274455	0.0107857	0.10	0.0228235	0.32
16	0.0298651	0.0063944	0.28	0.0156476	0.51
24	0.0302147	0.0044049	0.47	0.0131288	0.98

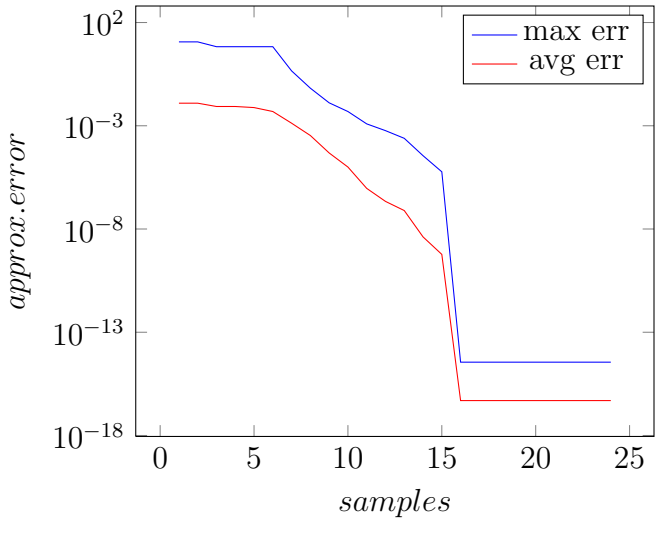


Figure 4.4: Approximation error over the number of samples for the intersection of random halfspaces with random polytopes with 16 facets.

faster if its optimal solution for the corresponding direction is close to the one computed in the last call.

Figure 4.4 shows the approximation error of the support function of the intersection with a halfspace as a function of the number of samples taken. We measure the absolute error over 10000 random instances of a polytope with 16 facets intersected with a halfspace. The polytope and the intersection are by construction non-empty and the halfspace is non-redundant. After 17 samples, both maximum and average error are below 10^{-13},

which is about as close as we expect given machine precision.

4.3 Intersecting a Set of Convex Sets with a Hyperplane/Halfspace

We are finally interested in computing the support function representation of the intersection of a flowpipe section with a guard set $\mathcal{G}$. A flowpipe section is a collection of convex sets and a guard set is a polyhedra, bounded or unbounded. In section 3.1, we described our algorithm to compute the support function of the intersection of a convex set with a hyperplane or halfspace . We can naively use the same algorithm for each and every convex set in the flowpipe collection to get a collection of support function represented intersection sets. This naive approach is expensive nevertheless.

To counteract the cost, we solve the minimization problems for each convex set in the flowpipe interval **simultaneously**. The underlying properties of the data structure used to represent the flowpipe are exploited to gain on time. Let us recall that we represent a flowpipe of size N given as $\Omega_0, \Omega_1, \ldots, \Omega_N$ as a $r \times N$ matrix which we call as the *Support Function Matrix* with $(i,j)^{th}$ entry denoting the support function sample of Ω_j in the direction l_i [FR09]. The SFM essentially provides a polyhedral approximation of each Ω_i in the r template directions. We use the algorithm in [GL08] to compute our SFM which for a given direction l, computes the support function of Ω_0 to Ω_N iteratively starting from Ω_0. Hence, for some flowpipe interval $\Omega_i, \ldots, \Omega_j$, if we need to compute the support function of Ω_k, $i \leq k \leq j$ in a new direction l, we actually compute the support function of atleast all the Ω_0 to Ω_k in the new direction l in our SFM. In our implementation, adding a new direction to a SFM is done through the *extend* operation. When we say extend an SFM in a direction l, the support function for all the N Ω's are added to the matrix in an additional row, N being the size of the flowpipe. This means that when we sample a convex set Ω_i of the flowpipe, we actually sample all the Ω's of the flowpipe in our SFM representation. Figure 4.5 shows the support function graphs of $sup_{\Omega_i}(l - \lambda n)$, i.e., in the λ domain where Ω_i is a convex set of the bouncing ball model flowpipe, n is the normal to the guard constraint in the model $(x \leq 0)$ and l is a given direction in which to compute the support function of the intersection set ($l = (0, 1)$ in this example). This figure shows the support function graphs for the 3 convex sets of the flowpipe at the second jump in the bouncing ball model that intersects with the guard. It shows that extending the sfm of the flowpipe in a direction corresponding to $\lambda = 5$ for example, samples all the three support functions of the three convex sets shown with the asterix mark. We use this fact to speed up our minima search when solving the minimization problems simultaneously.

For a k sized flowpipe interval and a guard constraint $g \in G$, we initialize k minimization problems. Each of the minimization problem then demands a new sample point. Any one of the demand is served and the SFM is extended in this new direction corresponding to the chosen sample point. After the SFM is extended, we know that each of the k functions in the minimization problem has been sampled as well in this new direction. We then call an *update_bounds* method which improves the lower and upper bound on the function minima, for each of the minimization problem. This is repeated iteratively until the minima is bounded in an interval with size less than a given tolerance value for

each of the minimization problem. The steps are shown in algorithm 4.2.

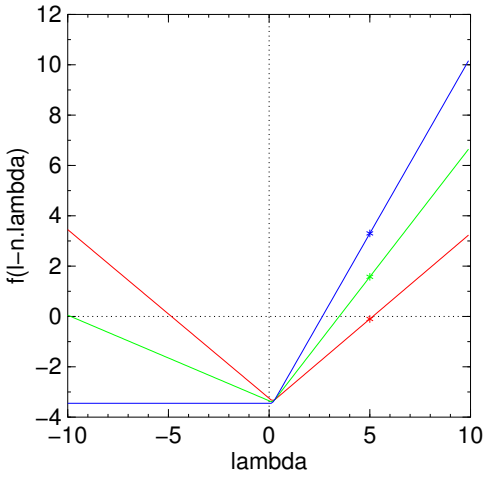


Figure 4.5: The three plots shows the support function graphs of three convex sets of a flowpipe of the bouncing ball model. Extending the sfm in a direction corresponding to $\lambda = 5$ samples all the three functions shown with asterix mark.

The *update_bounds* routine worth some illustration. This method takes the new sample, say λ and based on the current pivots, it calculates tighter bounds on the minima. If the λ is outside the minima containing interval, i.e., $\lambda < p_1$ or $\lambda > p_4$ or $\lambda > p_5$ in the five pivots state, then the method simply returns without updating. Also, if the λ is one of the pivots, then it is redundant and the method return without updating. Otherwise, based on the four possible positions of λ and the current state of the algorithm, the method improves the bounds on the function minima. Recall from the state machine in Figure 3.6 illustrating the Lower Bound Search algorithm in section 3.2.2 that state $S1$ denote $f(p_2) > f(p_3)$, state $S2$ denote $f(p_2) < f(p_3)$, state $S3$ denote $f(p_2) = f(p_3)$ and state $S4$ denote the 5 pivots state as mentioned in section 3.2.2.4. Therefore, the Lower Bound Search algorithm can be at one of these four states. Let us now discuss the actions on the four possible positions of the chosen λ in what follows:

1. $p_1 < \lambda < p_2$: Algorithm in state $S1$: The new sample is assigned as pivot p_1. By convexity of the function, $f(\lambda) > f(p_2)$ and hence the algorithm remains in state $S1$. The lower and upper bounds on the minima are recomputed as described in 3.2.2.1.

Algorithm in state $S2$: We compare $f(\lambda)$ with $f(p_2)$. if $f(\lambda) > f(p_2)$, λ is renamed to p_2, p_2 is renamed to p_3, p_3 is renamed to p_4 and p_4 is renamed to p_5. The algorithm moves to the state $S4$. The lower and upper bounds are recomputed as described in section 3.2.2.4. if $f(\lambda) = f(p_2)$, λ is renamed to p_2, p_2 is renamed to p_3 and p_3 is renamed to p_4. The algorithm moves to state $S3$ and the bounds on the minima are recomputed as described in section 3.2.2.2. If $f(\lambda) < f(p_2)$, λ is renamed to p_2, p_2 is renamed to p_3 and p_3 is renamed to p_4. The algorithm remains in state $S1$. The bounds are recomputed as described in section 3.2.2.1.

Algorithm in state $S3$: By convexity, $f(\lambda)$ can be either greater or equal to $f(p_2)$. If $f(\lambda) > f(p_2)$, λ is renamed to p_1, p_2 is renamed to p_3 and p_3 is renamed to p_4. The algorithm moves to state $S3$ and the bounds on the minima are recomputed as in section

3.2.2.2. If $f(\lambda) = f(p_2)$, we have $f(\lambda) = f(p_2) = f(p_3)$. By corollary 3.1, we have the function minima at $f(\lambda)$ and the algorithm terminates.

Algorithm in state $S4$: By convexity, the only possibility is $f(\lambda) > f(p_2)$. λ is renamed to p_1 and the algorithm remains in state $S4$. The bounds on the minima are recomputed as in section 3.2.2.4.

2. $p_2 < \lambda < p_3$: Algorithm in state $S1$: If $f(\lambda) > f(p_3)$, then p_2 is renamed to p_1 and λ is renamed to p_2. The algorithm remains in state $S1$ and the bounds on the minima are recomputed as in 3.2.2.1. If $f(\lambda) = f(p_3)$, then p_2 is renamed to p_1 and λ is renamed to p_2. The algorithm changes to state $S3$ and the bounds on the minima are recomputed as in 3.2.2.2. if $f(\lambda) < f(p_3)$ then λ is renamed to p_3, p_3 is renamed to p_4 and p_4 is renamed to p_5. The algorithm moves the state $S4$ and the bounds on the minima are recomputed as in section 3.2.2.4.

Algorithm in state $S2$: This is symmetrical to the previous with different pivot renamings. If $f(\lambda) > f(p_3)$, then p_3 is renamed to p_4 and λ is renamed to p_3. The algorithm remains in state $S2$ and the bounds on the minima are recomputed as in 3.2.2.3. If $f(\lambda) = f(p_3)$, then p_3 is renamed to p_4 and λ is renamed to p_3. The algorithm changes to state $S3$ and the bounds on the minima are recomputed as in 3.2.2.2. if $f(\lambda) < f(p_3)$ then λ is renamed to p_3, p_3 is renamed to p_4 and p_4 is renamed to p_5. The algorithm moves the state $S4$ and the bounds on the minima are recomputed as in section 3.2.2.4.

Algorithm in state $S3$: By convexity, $f(\lambda)$ can be either greater or equal to $f(p_3)$. If $f(\lambda) = f(p_3)$, then by corollary 3.1, minima is at $f(\lambda)$ and the algorithm terminates. if $f(\lambda) < f(p_3)$ then λ is renamed to p_3, p_3 is renamed to p_4 and p_4 is renamed to p_5. The algorithm moves to state $S4$ and the bounds on the minima are recomputed as in 3.2.2.4.

Algorithm in state $S4$: If $f(\lambda) > f(p_3)$ then p_2 is renamed to p_1 and λ is renamed to p_2. The algorithm remains in state $S4$ and the bounds on the minima are recomputed as in section 3.2.2.4. If $f(\lambda) = f(p_3)$ then p_2 is renamed to p_1 and λ is renamed to p_2. The algorithm moves to state $S3$ and the bounds on the minima are recomputed as in section 3.2.2.2. If $f(\lambda) < f(p_3)$ then λ is renamed to p_3, p_3 is renamed to p_4 and p_4 is renamed to p_5. The algorithm remains in state $S4$ and the bounds on the minima are recomputed as in section 3.2.2.4.

3. $p_3 < \lambda < p_4$: Algorithm in state $S1$: if $f(\lambda) > f(p_4)$ then we rename λ to p_4 and p_4 is renamed to p_5. The algorithm moves to state $S4$ and the bounds on the minima are recomputed as in 3.2.2.4. If $f(\lambda) = f(p_3)$ then p_2 is renamed to p_1, p_3 is renamed to p_2 and λ is renamed to p_3. The algorithm moves to state $S3$ and the bounds on the minima are recomputed as in 3.2.2.2. If $f(\lambda) < f(p_3)$ then p_2 is renamed to p_1, p_3 is renamed to p_2 and λ is renamed to p_3. The algorithm remain in state $S1$ and the bounds on the minima are recomputed as in section 3.2.2.1.

Algorithm in state $S2$: By convexity, the only possibility is $f(\lambda) > f(p_3)$. λ is renamed to p_4 and the algorithm remains in state $S2$. The bounds on the minima are recomputed as in 3.2.2.3.

Algorithm in state $S3$: By convexity, the only possibility is $f(\lambda) > f(p_3)$. λ is renamed to p_4 and the algorithm remains in state $S3$. The bounds on the minima are recomputed as in 3.2.2.2.

Algorithm in state $S4$: If $f(\lambda) > f(p_3)$ then we rename λ to p_4 and p_4 ot p_5. The algorithm remains in state $S4$ and the bounds on the minima are recomputed as in 3.2.2.4.

If $f(\lambda) = f(p_3)$ then we rename p_2 to p_1, p_3 to p_2 and λ to p_3. The algorithm moves to state $S3$ and the bounds on the minima are recomputed as in 3.2.2.2. If $f(\lambda) < f(p_3)$ then we rename p_2 to p_1, p_3 to p_2 and λ to p_3. The algorithm remains in state $S4$ and the bounds on the minima are recomputed as in section 3.2.2.4.

4. $p_4 < \lambda < p_5$: This case is interesting only when the algorithm is in state $S4$, i.e., the 5 pivots state. By convexity, the only possibility is $f(\lambda) > f(p_4)$. We rename λ to p_5 and the algorithm remains in state $S4$. The bounds on the function minima are recomputed as in section 3.2.2.4.

Algorithm 4.2 Simultaneous solving of one dimensional minima search problems

Require: Functions $[f^i, f^j]$ for each convex set of the flowpipe interval $[\Omega_i, \Omega_j]$ and $tol \in \mathbb{R}$
Ensure: $[l_k, u_k]$ such that $l_k < f^k_{\min} < u_k$ and $u_k - l_k <= tol$, $\forall k \in [i, j]$

```
1: stop ← false
2: list l of requested sampling points = ∅
3:
4: while !stop do
5:    for k = i → j do
6:       opt_prb_k ← init_problem(f^k)
7:       is_active_k ← true
8:       opt_prb_k.minbrak() {brackets the function minima with four pivots.}
9:       l.push(opt_prb_k.get_sampling_point()) {Each problem requests a new sampling point.}
10:   end for
11:   s ← l.choose_point() {One of the requested sampling point is selected.}
12:   for k = i → j do
13:      if is_active_k then
14:         opt_prb_k.update_bounds(s)
15:         [l_k, u_k] ← opt_prb_k.get_bounds()
16:         if u_k − l_k < tol then
17:            is_active_k ← false
18:         end if
19:      end if
20:   end for
21:   if ∀k ∈ [i, j], u_k − l_k < tol then
22:      stop ← true
23:   end if
24: end while
```

4.3.1 Convex Hull of the Intersection

We mentioned earlier that there could be many convex sets of the flowpipe which intersect with the guard set and if we treat them individually, we could have a large number of initial sets to begin the time elapse operation in the target location. The notion of clustering discussed earlier in section 2.4 showed a way of reducing the number of convex sets using template hull or convex hull clustering or both.

With convex hull for example, we could think of the following two approaches:

2.1 We compute the convex hull of the union of Ω_i's of the intersecting interval and then find its intersection with the guard set,

$$S_{\mathcal{G}}^{\mathcal{I}} = \mathrm{CH}\Big(\bigcup_{i_{\min} \le i \le i_{\max}} \big(\Omega_i\big)\Big) \cap \mathcal{G} \tag{4.9}$$

2.2 We compute the intersection first for each Ω_i with the guard set $\mathcal{G}$ and then compute the convex hull of the union of the results,

$$S_{\mathcal{G}}^{\mathcal{I}} = \mathrm{CH}\Big(\bigcup_{i_{\min} \le i \le i_{\max}} \big(\Omega_i \cap \mathcal{G}\big)\Big) \tag{4.10}$$

Let us now discuss the complexity of the above two approaches for support function representation and polytope representation of Ω_i. We are interested in two set operations, namely, intersection and convex hull of the union of sets. For Ω_i's represented as H-polytopes, computing the convex hull of their union is an expensive operation [Tiw08] but the intersection operation with a polyhedral guard set given as H-polytope is an easy operation (If redundant constraints are acceptable).

For Ω_i represented by the support function, computing the support function representation of the convex hull of their union is an easy operation but computing the support function representation of the intersection is expensive.

We discussed the idea of applying convex hull to the convex sets before and after computing the intersection in (4.9) and (4.10) respectively. Intuitively, (4.10) is expected to return more precise intersection compared to (4.9). In this section, we show how we solve (4.10) using our minimization algorithm. There are essentially two approaches which we consider.

- We solve the k minimization problems for the k intersecting flowpipe section members simultaneously as described in section 4.3 and then take the max of the computed mins as the support function value by property (2.17) of support functions.
- We solve the k minimization problems simultaneously with **branch and bound** method [LD60]. We describe this approach below.

Likewise in the previous section, we solve the minimization problem for each of the convex set of the flowpipe intersecting with the guard simultaneously. For a flowpipe interval $[\Omega_i, \Omega_j]$ whose intersection with the guard set $\mathcal{G}$ is not empty, we are interested in computing the support function representation of S_G^I as defined in (4.10) where I is an interval of indices of the flowpipe, $[i, j]$ in this case and $\mathcal{G}$ is a polyhedral guard set. We shall illustrate the algorithm for intersection with a single constraint which can be extended to a list of constraints of polyhedral $\mathcal{G}$ as shown in section 4.4.

Using (2.17), we have the following relation for support function of S_G^I:

$$sup_{S_G^I}(l) = \max\{sup_{\Omega_i \cap G}(l), \ldots, sup_{\Omega_j \cap G}(l)\} \tag{4.11}$$

To solve (4.10), we modify our algorithm 4.2 presented in the previous section with additional **branch and bound** technique. As we solve the k minimization problems

Algorithm 4.3 Computing the max of mins while solving the minimization problems simultaneously

Require: Functions $[f^i, f^j]$ for each convex set of the flowpipe interval $[\Omega_i, \Omega_j]$ and $tol \in \mathbb{R}$
Ensure: $[l_{\max}, u_{\max}]$ such that $l_{\max} < \max\{f^i_{\min}, \ldots, f^j_{\min}\} < u_{\max}$ and $u_{\max} - l_{\max} <= tol$.

stop $\leftarrow$ false
list l of sample points.
Boolean vector *active* {This vector keeps track of the discarded set of problems}
$l_{\max} \leftarrow -\infty$
while !stop **do**
  for $k = i \rightarrow j$ **do**
    $opt\_prb_k \leftarrow init\_problem(f^k)$
    $opt\_prb_k.minbrak()$ {brackets the function minima with 4 pivots}
    $active[j] \leftarrow true$ {Initially all the problems are active}
    $l.push\_back(opt\_prb_k.get\_next\_sample())$
  end for
  $s \leftarrow l.choose\_sample()$
  $u_{\max} \leftarrow -\infty$
  for $k = i \rightarrow j$ **do**
    if $active[k]$ **then**
      $opt\_prb_k.update\_bounds(s)$
      $[l_k, u_k] \leftarrow opt\_prb_k.get\_bounds()$
      if $u_k < l_{\max}$ **then**
        $active[k] \leftarrow false$ {Discarding the problem}
      end if
      if $l_k > l_{\max}$ **then**
        $l_{\max} \leftarrow l_k$
      end if
      if $u_k > u_{\max}$ **then**
        $u_{\max} \leftarrow u_k$
      end if
    end if
  end for
  if $\forall k \in [i, j]$ & $active[k]$, $u_k - l_k < tol$ **then**
    $stop \leftarrow true$
  end if
end while

simultaneously, k being the size of the intersection flowpipe interval I, each of the minimization problem updates its lower and upper bound on the minima iteratively. Since we are interested in computing the max of the mins, we compute the max of the upper and lower bounds of all the minimization problems at every iteration. If the upper bound on the minima of a minimization problem is less than the max of the lower bound computed for all the problems, then the problem is *discarded* for further computation since we know that it is is not going to contribute to the final result (branch and bound). We discard as many problems as we can at each iteration and stop until the difference between the upper and the lower bound is less than a given tolerance value for all the remaining problems. The maximum upper bound of all the problems that remains at the end is returned. The algorithm is illustrated in 4.3.

Algorithm 4.4

Require: Flowpipe interval $[\Omega_i, \Omega_j]$ of size N, a split size d and a polyhedral guard set G.
Ensure: Collection of convex sets $S_G^{I^k}$, where $1 \leq k \leq ceil(N/d)$.

```
for k = 0 → ceil(N/d) − 1 do
    l ← i + d ∗ k
    u ← low + d − 1
    S_G^{I^k} ← chull(Ω_l ∩ G, ..., Ω_u ∩ G)
end for
```

4.3.2 Convex Hull with Flowpipe Interval Splitting

Applying convex hull to the union of the intersection of the convex sets of the flowpipe with the guard set solves the problem of having numerous initial sets at the next location after a discrete transition, but it brings in larger over-approximation error. To trade-off between the over-approximation error and the speed of computation, we introduce the idea of what we call *splitting* the flowpipe interval before intersection. A user can supply a split size d, where d should be at most the size of the flowpipe interval that intersects with the guard set. What it means is that the flowpipe interval is split into smaller intervals of size at most d and then we compute the convex hull of the union of the intersection of the convex sets of these smaller intervals with the guard set. Hence, for a flowpipe interval of size N with a split number d, we are expected to get $ceil(N/d)$ convex sets as the result. Larger the split size, lesser will be the resulting number of convex sets and hence larger the over-approximation error but faster the computation. Convex hull with splitting is illustrated as pseudo code in Algorithm 4.4.

4.4 Intersecting a Set of Convex Sets with a Polyhedron

We assume in this section that the polyhedron is given to us in H representation, i.e., as an intersection of halfspaces. We first take a look at the intersection of a single convex set $\mathcal{X}$ with a polyhedron $\mathcal{P}$. Since $\mathcal{P}$ is an intersection of halfspaces, we can apply lemma 3.3 repeatedly to obtain the support function of its intersection with a convex set $\mathcal{X}$:

Lemma 4.3. $sup_{\mathcal{X}\cap\mathcal{P}}(l) = \inf_{\lambda\in\mathbb{R}^m,\lambda\geq 0} sup_{\mathcal{X}}\left(\ell - \sum_i \lambda_i n_i\right) + \sum_i \lambda_i \gamma_i$

This is a convex optimization problem over m variables where m is the number of constraints in $\mathcal{P}$.

In our implementation, we compute the intersection with each halfspace separately. We intersect $\mathcal{X}$ with each halfspace of $\mathcal{P}$ separately and combine the results with the following approximation

$$sup_{\mathcal{X}\cap\mathcal{P}}(\ell) \leq \min_{i=1,\ldots,m} sup_{\mathcal{X}\cap\{n.x\leq\gamma\}}(\ell) \tag{4.12}$$

For the intersection of a collection of convex sets with a polyhedron $\mathcal{P}$, we take the convex hull of the intersection of each Ω_i of the flowpipe section $\mathcal{I}_k$ that intersects with $\mathcal{P}$. Each

halfspace of $\mathcal{P}$ is considered separately. Therefore, for the intersection of the flowpipe Ω_i with the j^{th} halfspace of $\mathcal{P}$, we must minimize:

$$f_j^i(\lambda) = sup_{\Omega_i}(\ell - \lambda n_j) + \lambda\gamma_j. \tag{4.13}$$

Applying the same approximation for polyhedron approximation as in 4.12, we obtain the approximation:

$$sup_{\mathcal{Y}_k}(\ell) = \max_{i\in\mathcal{I}_k} \min_{j=1,\ldots,m} \inf_{\lambda\in\mathbb{R}^{\geq 0}} f_j^i(\lambda). \tag{4.14}$$

As with 4.11 we can use a *branch and bound* algorithm to eliminate the instances of i, j for which the upper bound of $f_k^i(\lambda)$ is lower than the largest of the lower bounds.

4.5 Computational Optimization

The computation of the sequence Ω_i amounts to a symbolic integration of the ODE (2.20), so the support function values of Ω_i depend on the support function values of Ω_{i-1}, etc. This gives us the following limitation, which will become an important when we consider intersections:

Assumption 4.1. *To compute* $\sup_{\Omega_i} \ell$*, we also need to compute* $\sup_{\Omega_j} \ell$ *for* $j = 0, \ldots, i-1$.

There is a partial remedy to this problem. Consider the case where we are interested in computing a subsequence of the flowpipe approximation Ω_i, say for $i \in [c, d]$. Under Assumption 4.1 this requires us to compute the $d+1$ sets with $i \in [0, d]$. We can reduce this computation burden as follows. The sequence Ω_i is constructed such that each set covers the flowpipe over a known time interval $[t_i, t_{i+1}]$. We decomposing the system into its autonomous dynamics ($\mathcal{U} = \emptyset$) and its input dynamics ($\mathcal{X} = \emptyset$). Recall that for autonomous dynamics, the set of states reached at exactly time t_c is $\mathcal{X}_{t_c} = e^{At_c}\mathcal{X}$. Starting the flowpipe computation for the autonomous dynamics from $t = t_c$ instead of $t = 0$, we end up with fewer sets to compute. Let

$$(\Omega_c^x, \ldots, \Omega_d^x) = \text{post}_c e^{At_c}\mathcal{X}, \emptyset, \tag{4.15}$$

$$(\Omega_0^u, \ldots, \Omega_c^u, \ldots, \Omega_d^u) = \text{post}_c \emptyset, \mathcal{U}, \tag{4.16}$$

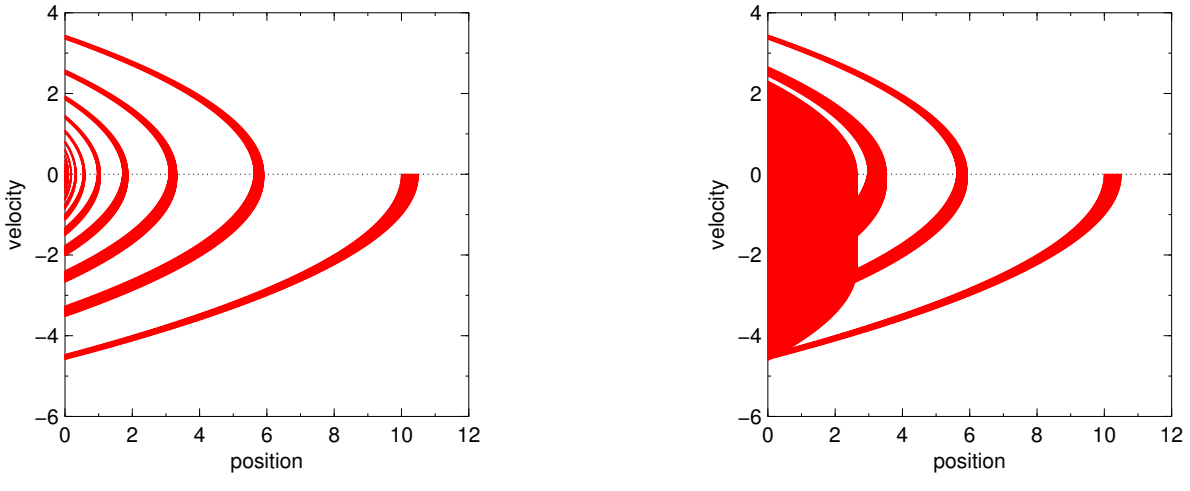
such that Ω_i^x and Ω_i^u cover the respective flowpipe on the same time interval $[t_i, t_{i+1}]$. Then using the superposition principle we have that $\Omega_i^x \oplus \Omega_i^u$ covers the flowpipe of $\mathcal{X}$ and $\mathcal{U}$ on the time interval $[t_i, t_{i+1}]$. This means we only need to compute the $d - c + 1$ values of (4.15). While (4.16) still requires the computation of $d + 1$ values, the set $\mathcal{U}$ is in practice often simple, e.g., a hyperbox, so that its support function can be computed much quicker than that of $\mathcal{X}$.

In our minimization algorithm to compute the support function of the flowpipe-guard intersection set, we need to sample the Ω_i frequently in new directions. The above observation largely reduces the computation overhead.

4.6 Case Studies

We illustrate the difference of our improved discrete image operator (2.32) and the standard discrete image operator (2.30) in terms of precision and computation time, on some case studies. We also compare our approach of computing the convex hull of the convex sets covering the flowpipe before intersecting with the guard set using the branch and bound method illustrated in section 4.3.1. In the comparisons below, we refer our new discrete image operator given in (2.32) as LBS intersection. LBS here stands for Lower Bound Search algorithm that we principally use to precisely compute the intersection of support function represented sets with polyhedral guard sets. Regarding clustering, we cluster the convex sets before (-) or after (+) we compute the image of the discrete transition. We consider as alternatives the template hull of all sets (TH), the convex hull of all sets (CH), and a mix of both (template hull of about 30%, then convex hull). 30% here means the percent of *clustering factor* described in the *clustering* part of section 2.4.

The case studies we use to illustrate are the simple bouncing ball model, the filtered oscillator and the colliding pendulum model. We also test our method on the navigation benchmark model given in [FI04] but do not make comparisons.



(a) Reachability upto fixpoint with LBS intersection

(b) Reachable set diverges with standard discrete image operation after the 5th jump.

Figure 4.6: Reachability up to fixpoint with LBS intersection which is not possible with standard discrete image operation.

Bouncing Ball The bouncing ball model consists of a hybrid automata with a single location having one and only self transition. The system has two variables namely the position x and the velocity v of the bouncing ball. The flow equation in the location is given by $\dot{x} = v$ and $\dot{v} = -g$, where g is a constant set to 1. The location invariant is $x \geq 0$. The self transition has the guard $x \leq 0 \wedge v < 0$ and an assignment $v' = -c.v$, where c is a constant set to 0.75. The constant c accounts for the damping effect during the jumps. Figure 4.6(a) shows the computed reachable set with LBS intersection along with convex hull clustering with branch and bound as described in section 4.3.1. Notice that here, the convex hull clustering is done before computing the assignment map (denoted by CH^{-}). With a time step of 0.001s and box directions, the fixpoint is reached after 22 jumps in 0.909s. With the same time step and directions, the standard discrete image

operator with either template or convex hull clustering does not reach fixpoint. The error is so large that the 5th jump reaches higher than the 4th and further jumps takes the reachable set to diverge to infinity. Figure 4.6(b) shows the result on 5 jumps with standard discrete post with template hull clustering.

Table 4.3 shows the time and precision comparison of the proposed discrete image computation with LBS intersection and the standard discrete image operation with clustering. We make comparison with the different variants of clustering on the standard discrete image operation as they affect the precision and computation time of the reachable set. We compute the reachable set for 5 jumps of the ball. The time step δ for the experiments is taken to be 0.025. For precision comparison, we consider the absolute difference between the empirical height of the 5th jump and the exact height and compute the percent error, i.e, percent error in height $= |(h_{empirical} - h)/h|$. A close approximation of the exact height is computed by running the reachability algorithm on support functions with very small time step δ (0.001) and many directions (uni32). The percent error in height is shown in the last column of table 4.3.

Table 4.3: Speed versus accuracy comparison of different variants of the discrete image computation, applied to the bouncing ball example. The accuracy shows in the percent error of the height of the 5th jump

direction	err	clustering	runtime(s)	percent err
standard discrete image computation				
box		TH^+	0.325	109.045
box		$TH\&CH^+$	0.613	109.045
box		CH^+	2.408	109.045
oct		TH^+	0.408	12.8319
oct		$TH\&CH^+$	0.625	12.8319
oct		CH^+	0.935	12.8319
discrete image with LBS intersection				
box	0.0	TH^+	0.198	0.35604
box	0.0	$TH\&CH^+$	0.214	0.35604
box	0.0	CH^+	0.216	0.35604
oct	0.0	TH^+	0.391	0.055
oct	0.0	$TH\&CH^+$	0.394	0.055
oct	0.0	CH^+	0.392	0.055
oct	0.01	TH^+	0.381	0.152
oct	0.1	TH^+	0.382	0.633
LBS intersection with convex hull clustering				
box	1.0	CH^-	0.2	2.556
box	0.2	CH^-	0.2	2.556
box	0.1	CH^-	0.2	2.556
box	0.01	CH^-	0.195	0.511
box	0.0	CH^-	0.196	0.35604
oct	1.0	CH^-	0.382	1.12179
oct	0.1	CH^-	0.382	1.12179
oct	0.0	CH^-	0.385	0.098

In the bouncing ball model, we see that the different clustering options does not make a difference in terms of precision. This is because very few flowpipe segments intersect with the guard constraint with the chosen time step. The best precision obtained is 0.055% error with our LBS intersection. The next best is 0.098% error with LBS intersection operation with convex hull clustering using branch and bound. Also notice the difference in percent error with varying error parameter (column 2) with our precise LBS intersection using branch and bound. The error value here signifies the intersection error tolerance value when computing the support function of the flowpipe-guard intersection set with the novel sandwich algorithm. Also observe that the percent error does not increase above a threshold on increasing the error paramater value arbitrarily. This is because our minima bracketing algorithm (see section 3.2.1) computes an optimal gap on the exact support function and this optimal gap computed by the minima bracketing routine is the maximum intersection error that could be tolerated. Hence, specifying a error value larger than this threshold will have no effect.

We also make comparison with a timed bouncing ball model which is constructed by adding an additional time variable with dynamics $\dot{t} = 1$ in the flow equation of the location. In the transition assignment, the time variable is not changed, i.e., $t' = t$. The additional time variable makes the timed bouncing ball a three dimensional system. The experiments are performed with a time step $\delta = 0.01$. Due to a small time step, more flowpipe segments intersect with the guard constraint and thus we see the effect of different clustering parameters. The precision and time comparison with the precise LBS intersection method is shown in table 4.4. The height of the 5th jump is taken for precision comparison because with large error in the discrete image computation, the height of the later jumps goes higher than the height of the previous jumps and diverges to infinity.

Table 4.4: Speed versus accuracy comparison of different variants of the discrete image computation, applied to the timed bouncing ball example. The accuracy shows in the height of the 5th jump

direction	err	clustering	runtime(s)	height
standard discrete image computation				
box		TH^+	1.3	3.054
box		$TH\&CH^+$	2.6	2.209
box		CH^+	31.2	2.016
oct		TH^+	3.0	0.972
oct		$TH\&CH^+$	12.7	0.901
oct		CH^+	36.6	0.844
discrete image with LBS intersection				
box	0.0	TH^+	1.3	1.080
box	0.0	$TH\&CH^+$	3.4	1.017
box	0.0	CH^+	55.9	0.904
LBS intersection with convex hull clustering				
box	1.0	CH^-	0.8	1.175
box	0.1	CH^-	0.6	0.815
box	0.0	CH^-	0.6	0.807

Filtered Oscillator A filtered oscillator is a switched oscillator system with a series of first order filters to the output x of the oscillator. The filters smooth x, producing a signal z whose amplitude diminishes as the number of filters increase. The oscillator is an affine system with variables x, y that switches between two equilibria in order to maintain a stable oscillation, which together with k filters yields a parameterized system with $k+2$ continuous variables. The hybrid automaton model of the oscillator and the filter are shown in Figure 4.7 and Figure 4.8 respectively. In the hybrid automaton model of the oscillator, a_1, a_2, x_0, y_0 and c are constants. For our experiment, we take $a_1 = -2$, $a_2 = -1$, $x_0 = y_0 = 0.7$ and $c = 0.5$. The invariant of location np, pp, pn and nn is given by $x \leq 0 \wedge y \geq (-c/x_0)x$, $x \geq 0 \wedge y \geq (-c/x_0)x$, $x \geq 0 \wedge y \leq (-c/x_0)x$ and $x \leq 0 \wedge y \leq (-c/x_0)x$ respectively. Notice that there is no guard constraint and assignment map over the transitions. This means that the guard set is true and the assignment map is identity. All the transitions have the synchronization label *hop*.

The filter model is simply a hybrid automaton with a single location. There is no constraint over the location invariant, i.e., the location invariant is true. The variable u is the input variable of the filter and x is the controlled output variable. c is a constant taken to be -5.

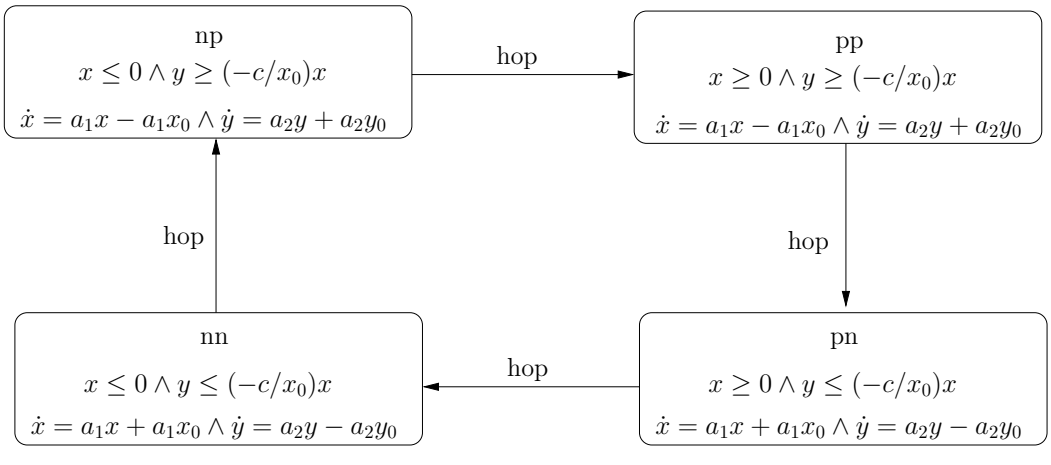


Figure 4.7: Hybrid Automaton Model of the Switched Oscillator

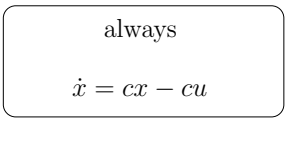


Figure 4.8: Hybrid Automaton Model of the Filter

Table 4.5: Speed versus accuracy comparison of different variants of the discrete image computation, for computing a fixed-point of the filtered oscillator example. The accuracy shows in the max amplitude of the output signal z

vars	δ	ε	clustering	runtime(s)	max. z	iter
standard discrete image computation						
6	0.01		TH^+	0.3	0.570	5
18	0.01		TH^+	2.1	0.361	9
34	0.01		TH^+	8.7	0.243	13
66	0.05		TH^+	17.4	0.291	23
130	0.05		TH^+	132.7	0.569	39
130	0.025		TH^+	206.0	0.166	41
precise intersection of convex hull with branch & bound						
6	0.01	0	CH^-	0.4	0.567	5
18	0.01	0	CH^-	2.4	0.356	9
34	0.01	0	CH^-	9.0	0.237	14
66	0.05	0.1	CH^-	17.3	0.243	23
66	0.05	0.01	CH^-	18.1	0.232	24
66	0.05	0.001	CH^-	27.4	0.192	37
66	0.05	0	CH^-	55.6	0.190	71
130	0.05	0.1	CH^-	126.0	0.339	39
130	0.05	0.01	CH^-	126.5	0.314	39
130	0.05	0.001	CH^-	205.5	0.190	39
130	0.025	0.01	CH^-	174.2	0.128	65

Table 4.5 shows results for up to 130 state variables, for both standard discrete image computation and the proposed variant with precise intersection. All instances are computed using box directions. The precise intersection variant outperforms the standard operator in precision, and often also in speed. In this example, the capacity to compute the intersection up to a given error (column 3) shows its benefits: a small but not too small error greatly reduces the analysis time, at an acceptable loss in accuracy.

Figure 4.9 shows the reachable set for a filtered oscillator model with 16 filters and hence having a total of 16+2 continuous variables.

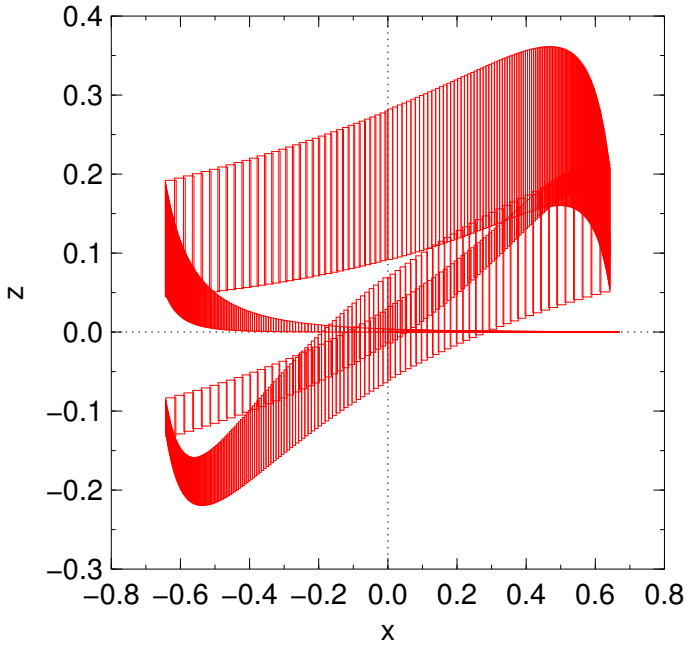


Figure 4.9: Reachability up to fixpoint computation for a 16th order filtered oscillator (18 vars) with LBS intersection routine.

Colliding pendulums This model consists of two pendulums of mass m and length ℓ such that they touch each other when at rest. For simplicity, the pendulums are considered to be point mass, i.e, their respective radius is 0. At rest, both the pendulums are at origin. To start the oscillation, one of the pendulums is taken some distance away from the origin and released. For our experiment, we displace the left pendulum from the origin to start the oscillation. This swinging pendulum then collides with the right pendulum at rest and transfers its momentum. The right pendulum swings and returns to collide again with the left pendulum and so on. The pendulum and the collision is modeled with two separate hybrid automata and they are composed. Both the hybrid automata consist of a single location and the system have five variables namely the displacement of the left pendulum x_l, velocity of the left pendulum v_l, displacement of the right pendulum x_r, velocity of the right pendulum v_r and time. The hybrid automaton model of the pendulum and the collision are shown in Figure 4.10 and Figure 4.11 respectively. The guard in the transition of the collision hybrid automata is given by $x_l == x_r \wedge v_l > v_r$ and the transition assignment is given by $v_l = e.v_r \wedge v_r = e.v_l$. e is the constant of elasticity and it is taken to be 0.95 . The continuous dynamics of the system is given by the flow equation in the location of the pendulum model. m, ℓ are the mass and length of the pendulum, taken as 0.05 and 3 respectively. g is the constant of gravity taken as 10. The time variable has the flow $\dot{t} = 1$ which is captured in another hybrid automata with only one location and is composed with the system.

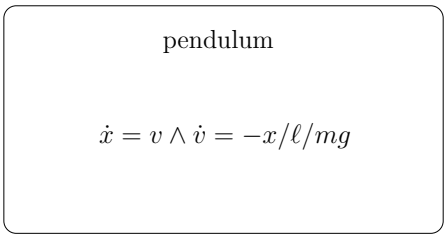


Figure 4.10: Hybrid Automaton Model of the Pendulum

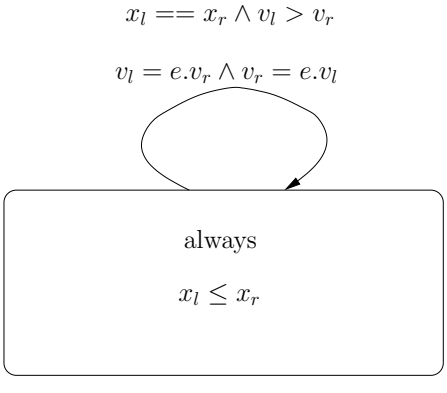
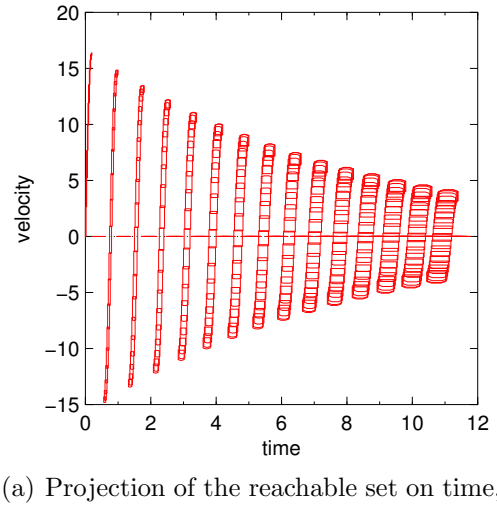


Figure 4.11: Hybrid Automaton Model of the Collision

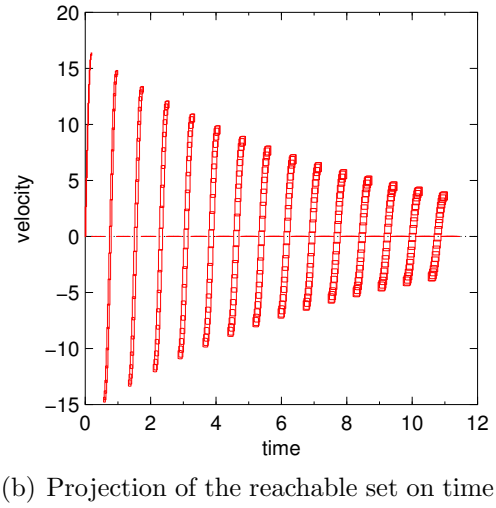
The speed and accuracy comparison is shown in table 4.6. In this model also, there is not much effect of the different clustering options over the accuracy because very few flowpipe sections (infact only 1) intersect with the guard constraint with the taken time step ($\delta = 0.025$). Also, it is observed that for this model LBS intersection lags behind in terms of computation time but precision-wise, LBS intersection outperforms the standard discrete image computation.

Table 4.6: Speed versus accuracy comparison of different variants of the discrete image computation, applied to the Colliding Pendulum example. The accuracy shows in the percent error in the maximum displacement of the left pendulum after the 28th collision.

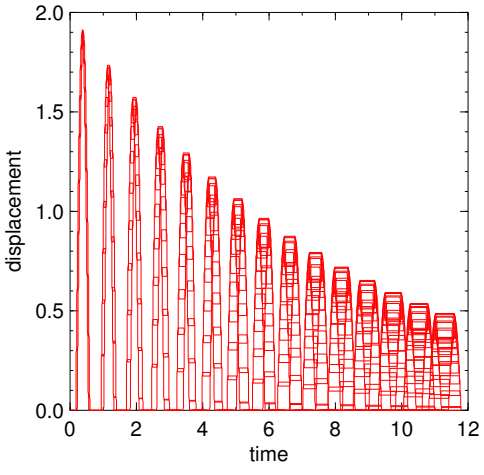
direction	err	clustering	runtime(s)	percent err
standard discrete image computation				
box		TH^+	0.885	15.391
box		$TH\&CH^+$	0.884	15.391
box		CH^+	0.907	15.391
oct		TH^+	2.682	15.391
oct		$TH\&CH^+$	2.683	15.391
oct		CH^+	2.672	15.391
discrete image with LBS intersection				
oct	0.0	TH^+	6.438	10.090
oct	0.0	$TH\&CH^+$	6.421	10.090
oct	0.0	CH^+	6.413	10.090
oct	0.01	TH^+	6.029	10.686
oct	0.05	TH^+	6.074	11.360
oct	0.1	TH^+	6.08	11.360
LBS intersection with convex hull clustering using branch & bound				
oct	0.0	CH^-	6.369	10.140
oct	0.01	CH^-	6.032	10.691
oct	0.02	CH^-	6.088	11.339
oct	0.05	CH^-	6.1	11.360
oct	0.10	CH^-	6.076	11.360
oct	1	CH^-	6.078	11.360



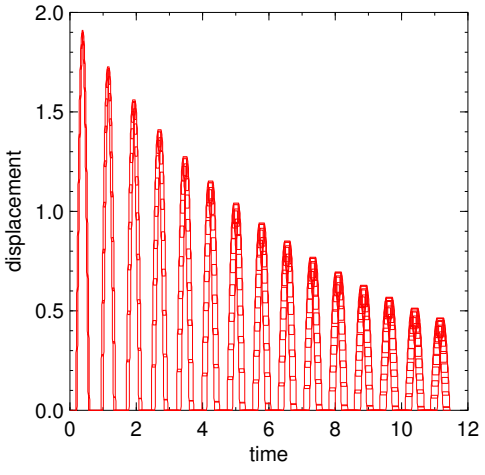
(a) Projection of the reachable set on time, velocity variables using standard discrete image computation



(b) Projection of the reachable set on time, velocity variables using precise discrete image computation (LBS)



(c) Projection of the reachable set on time, displacement variables using standard discrete image computation



(d) Projection of the reachable set on time, displacement variables using precise discrete image computation (LBS)

Figure 4.12: Illustrating the precision in the computed reachable set with LBS intersection. Notice the error accumulation with collisions with the standard discrete image computation.

Navigation Benchmark The navigation benchmark presented in [FI04] models the motion of an object in $\mathbb{R}^2$ plane. The plane in which the object can move is partitioned into an $n \times m$ grid and each cell of the grid has a designated desired velocity v_d. The actual velocity of the moving body is given by the differential equation $\dot{v} = A(v - v_d)$, A being a 2×2 matrix. The reader is referred to [FI04] for more details. Different instances of this model is provided by the author in the website (http://www.cse.unsw.edu.au/ ansgar/benchmark/) and we run our algorithm on them, namely NAV01, NAV02, NAV04.

Fixpoint is not found for NAV01, NAV02, NAV04 model with the standard discrete image computation. It is also not found with the LBS intersection. Figure 4.13 shows the computed reach set with the LBS intersection routine with convex hull clustering on the NAV04 model.

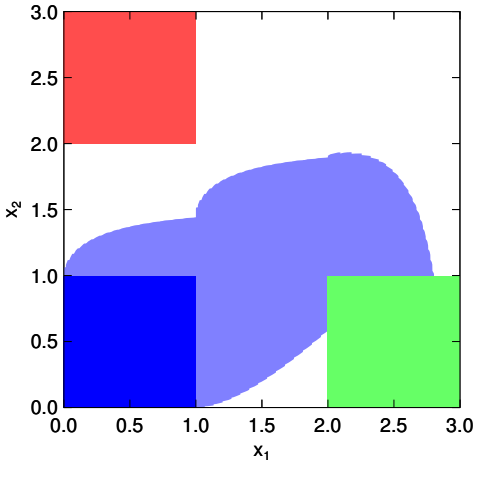


Figure 4.13: NAV04

Chapter 5

SpaceEx: A Tool Platform for Hybrid Systems Verification

SpaceEx is a tool platform for safety analysis of hybrid systems. It is so far the most scalable tool capable of handling hybrid systems with affine continuous dynamics with as many as 200 variables [FLGD+11]. SpaceEx is highly automated and analysis can be fine tuned using a number of parameter settings at the disposal of the user. SpaceEx consists of (1) The Analysis Core, (2) The Web-Interface and (3) A Model Editor. All three components of SpaceEx can be downloaded from the website [http://spaceex.imag.fr/].

The analysis core is a command line engine that takes the hybrid system specification under analysis in a XML based format which we call the *SX* format. The tunable parameters of the analysis could be either specified as command line options or in a configuration file. The core engine generates the corresponding output in the specified output file(s) in the specified format(s).

The model editor is a GUI based editor for specifying the hybrid system as a network of hybrid automata. The model is saved in a file in the SpaceEx's SX format.

The web interface is a GUI for running the analysis core over a web browser. The web interface calls the analysis core via a web server which may be running remotely or locally in a virtual machine.

In this chapter, we discuss about the *Analysis Core* of SpaceEx. It is implemented in the standard C++ programming language [Str86]. [1]

5.1 Requirements for an Extendable Tool Platform

Our goal is to enable the implementation of a number of different approaches to computing the set of reachable states using Alg. 2.2, as well as enabling their eventual combination and further enhancements.

We consider the following approaches for computing reachability and safety, which we find amenable to Alg. 2.2:

- Constant continuous and affine discrete dynamics

[1] This chapter contains excerpts from the document [Fre] and from the publication [FR09].

(A) HyTech [HHWT97]

(B) PHAVer [Fre08]

- Affine continuous and discrete dynamics

(C) d/dt [ADM02]

(D) Using zonotopes [Gir05]

(E) Using support functions [GL08]

(F) Algorithmic improvements to compute post_c for D,E [GGM06]

- Nonlinear dynamics and abstraction refinement

(G) Approximating nonlinear dynamics by hybridization [ADG03]

(H) Forward/backward refinement [FKR06]

(I) CEGAR-type approaches [FJK08]

The reachability techniques in A,B are for piecewise constant derivatives, exact as well as overapproximative over an infinite time horizon. In C, affine continuous and discrete dynamics are overapproximated by discretizing time over a finite time horizon. In D and E, this technique is improved by exploiting the advantages of a particular representation of continuous sets, plus some low-level algorithmic improvements. In G, the techniques for affine dynamics are extended to nonlinear dynamics by overapproximation based on partitioning the state space. In H, a very simple abstraction/refinement technique is used for deciding safety, and more sophisticated ones based on *counter example guided abstraction refinement* (CEGAR) can be found in I. Approaches A–E constitute low-level algorithms that deal with computing post-images for particular dynamics, while G–I are high-level techniques that use low-level reachability algorithms as an intermediate step in a larger scheme.

An analysis of common elements and differences shall provide us with the basis for our design.

5.1.1 Common Elements

The system under examination is described as a network of interacting automata. The specification consists of the set of initial and (for safety) forbidden states. In addition, the user has to provide analysis parameters such as discretization time steps or partition sizes. For the analysis, a parallel composition operator transforms the automaton network into a single automaton, possibly on the fly. The set of reachable states is computed using some variant of Alg. 2.1. The resulting set of states undergoes some basic processing (intersection with forbidden states, projection onto variables of interest), and is output to a file or visualized.

5.1.2 Differences

The approaches we consider differ along the following lines:

5.1.2.1 Set Representations

Polyhedra (A,B), zonotopes (D), and support functions (E) have each various advantages and disadvantages on fundamental set operations. For example, for polyhedra in constraint form computing intersection is cheap and Minkowski sum is expensive, while for zonotopes Minkowski sum is cheap and the intersection of two zonotopes is not generally a zonotope.

5.1.2.2 Discrete post-computations

Various exact as well as overapproximative techniques for computing the image of discrete transitions are available (such as taking the convex hull), depending on the set representation. Most techniques apply to discrete dynamics in the form of affine maps (resets). For example, the image of an affine map is cheap for zonotopes and polyhedra in generator form, but not for polyhedra in constraint form.

5.1.2.3 Continuous post-computations

Computing the image of a set after time elapse generally necessitates an overapproximations. Different techniques are applicable according to the type of continuous dynamics as well as the set representation. For A,B the image is over infinite time, while C,D,E,F discretize time and compute it over a bounded interval. Even for just linear dynamics, variations abound. For example, F avoids the wrapping effect by essentially reordering the computation and its approach is applicable to D,E.

5.1.2.4 State exploration

Most approaches are defined for forward reachability, but can equally be applied as backward reachability by reversing the system dynamics. One direction may work better than another depending on the characteristics of the system [Mit07], and H combines both. I requires keeping track of the dependency graph between symbolic states, i.e., which are the successor states of which. The explored states need to be stored in some form of passed/waiting list, and at each iteration the explored states need to be separated into those that are new and those that already been explored, which involves some form of difference operation (exact, overapproximative, see A).

5.1.2.5 Model transformations

Hybridization (G) and abstraction/refinement techniques (B,H,I) involve duplicating (splitting) locations, adding and removing transitions, and modifying dynamics and invariants. Such changes in the model must be compatible with the state exploration if they are to be carried out on the fly, or if state representations are to be compatible with different variants of the same model.

5.1.2.6 High level algorithms

In abstraction/refinement schemes like H or I, computing the reachable states is just one step in a larger process. They require certain low-level information like the dependency graph and counter examples to be accessible, and entail model transformations.

5.1.2.7 Automaton composition

Composition operators differ in the type of communication (synchronization) and how variables are shared (A versus B).

5.2 Design Specification

5.2.1 Principal Elements

Based on the survey and Alg. 2.2, we define the following *principal elements* and their operations:

- automaton representation : add locations, transitions
- automaton network representation (controls composition; itself an automaton) : add automata
- discrete and continuous set representations : inclusion and emptiness tests, transforms (intersection, affine maps, etc.)
- adapt: convert sets and dynamics to the right form (if possible)
- PWL : add, pop symbolic states
- continuous-post :transform a symbolic state into a set of symbolic states
- discrete-post : transform a symbolic state into a finite set of symbolic states

Implementation choices depend on each other. E.g., a specific continuous-post might only apply to affine dynamics and require polyhedra as set representations. At the same time, we would like to keep the concrete classes encapsulated as much as possible; whoever writes the polyhedron class may not know anything about hybrid automata.

This leads us to the following design principles:

- Implementations for the principal elements should be interchangeable.
- The principal elements should be used exclusively in Alg. 2.2 (instead of creating new algorithms that add elements or change the order); this shall guarantee that implementations from different sources remain interchangeable and as compatible as possible, avoiding divergence between different implementations.
- Compatability between principal elements is optional. We assume that anyone selecting a set of principal elements to create a scenario has expert knowledge. It suffices that an exception is created when an incompatibility is detected during execution.
- Low-level operations on continuous sets take up most of the computation time, so the overhead of polymorphism, operations on discrete sets etc. is considered negligible.

- The number of (convex) continuous sets created during exploration is large compared to the number of discrete sets, justifying additional effort, e.g., to compact sets of symbolic states.
- The number of different set representation is small and varies little compared to the other principal elements (post-operators, PWL). It is therefore acceptable that adding a new set representation requires updating the other principal elements (which is required for applying the visitor pattern in certain components).
- Advanced algorithms modify the system model (automaton network) on the fly or between re-runs of the reachability algorithm. Set representations need to be compatible with corresponding changes in locations and transitions, e.g., using keys to refer to previous versions of the location or transition.

5.2.2 Tool Architecture and Execution

We define for each of the principal elements an abstract base class, from which implementations must be derived. We call a set of implementations for the principal elements a *scenario implementation*, and define a scenario class to hold references to them, similar to the strategy design pattern. Given a scenario object, our implementation of Alg. 2.1 uses these references to instantiate automaton and set representation, and carry out operations on symbolic states and the PWL.

A run of the tool (assuming the model has already been generated possibly in a graphical editor) consists of the following steps, as shown in Fig. 5.1:

1. The user provides the input : models (XML), user commands, scenario selection, output selection.
2. The input file is parsed to generate a general representation of the automata (transitions/locations) and sets (initial states, bad states).
3. The general automata are adapted to the right set representation and dynamics according to the scenario (adapt).
4. The automaton network is instantiated according to the scenario.
5. The user selected algorithm (reachability, safety) is executed, using the elements provided by the scenario (PWL, post).
6. The output is created : visualization, file export (model, states).

User options are used to select the scenario, additional options can be passed directly to the scenario.

5.3 Tool Implementations

SpaceEx includes default, straightforward implementations for discrete sets, automata, automata networks and the PWL (linked list). For more details, the reader is referred

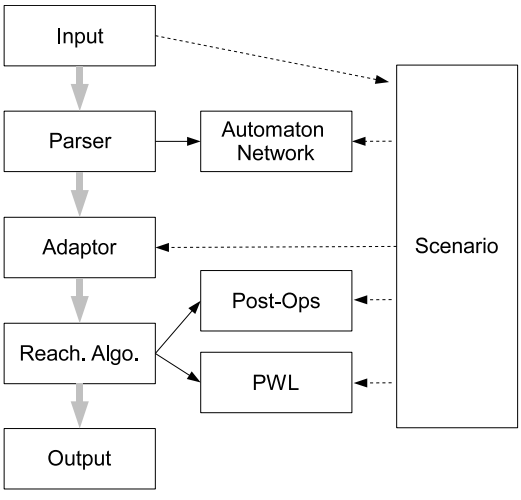


Figure 5.1: Schematic of the tool architecture (solid arrows represent acquaintance between objects, dashed arrows represent instantiation). Grey arrows indicate in which order the different components are executed

to a similar implementation in [Fre08]. Tool implementations that use these must only provide the remaining elements: representations of sets, dynamics, its adapters, and post-operators. The analysis core of SpaceEx currently implements two scenario for the reachable set computation, namely the PHAVer scenario and the LGG scenario. PHAVer scenario is for the reachability analysis of *linear hybrid* systems modeled with LHA. The algorithms used here are similar to that used in the tool PHAVer [Fre08]. It is to be noticed that hybrid systems with affine continuous dynamics cannot be run in the current implementation of the PHAVer scenario although the tool PHAVer run on them by approximating the affine dynamics with linear dynamics using state space partitioning.

The LGG scenario is for the analysis of hybrid systems having affine dynamics and non-deterministic inputs. LGG implements a variant of the support function based reachability algorithm given in [GL08]. The LGG scenario comes with a number of tunable parameters to be set by the user before initiating the reachable set computation.

Recently a *simulation* scenario has also been added to SpaceEx which generates simulation traces on the provided initial points. The simulation scenario is out of scope of this thesis.

Options can be set via the command line or via the web interface. The configuration files saved by the web interface can also be read directly by the command line tool. Note that it is possible to display the command line generated by the web interface, which may be useful for creating scripts etc.

We first present the general reachability algorithm in SpaceEx.

5.3.0.1 Reachability Algorithm

The reachability algorithm using symbolic states presented in section 2.1 in chapter 2 is executed. Recall that reachability for hybrid automata is undecidable in general, and this procedure is not guaranteed to terminate. Upon termination, the result is an overapproximation of the reachable states.

The following options are available to control the reachability algorithm:

- **Max. iterations**: Maximum number of iterations for the reachability algorithm, which is the total number of discrete post computations on symbolic states. If negative, the algorithm terminates only when a fixed point is reached.
- **Relative and absolute error**: These values are used when comparing floating point values and deciding whether they are considered equal. This impacts mainly tests for containment and emptiness of objects.
- **Merging passed with waiting list**: When a new state A contains a state B already on the passed list, B is by default replaced by A on the passed list. This merging process incurs the cost of containment checking and can be disabled.

5.3.1 Phaver Scenario

The Phaver scenario is for LHA models. A *linear hybrid automaton* (LHA) is a hybrid automaton whose continuous sets and relations are given by convex linear constraints over, respectively, the variables (invariant, initial states), the derivatives (flow), and the variables distinguishing before and after a jump (jump relation). This means that the continuous dynamics are nondeterministic with constant bounds, e.g., $1 \leq \dot{x} \leq 2$ or $\dot{x}+\dot{y}=0$. The discrete dynamics are nondeterministic affine, e.g., $x'=0$ or $x'=a*x+b$.

We represent continuous sets as polyhedra and provide a straightforward implementation based on linear programming to decide containment and emptiness. Fourier-Motzkin elimination is used for existential quantification. A generic lp-solver interface allows us to use different linear programming solvers, such as the GLPK [Mak09].

The continuous dynamics are modeled as the continuous set of derivatives for each location. The discrete dynamics (jump relations) are modeled as a discrete set over primed variables (after the jump) and unprimed variables (before the jump).

For LHA, the post-operators are first-order predicates whose solutions can be computed using the above standard operations on polyhedra.

5.3.2 Support Function Scenario

For affine continuous and discrete dynamics, an efficient approach to compute the reachable states has been proposed in [GL08]. The continuous dynamics is of the form (2.20) and the discrete dynamics is of the form given in (1.2). Given a set of directions, it uses polyhedral over-approximations, where each face of the polyhedron is a tight bound on the original set in one of the given directions.

To make the approach scalable, the support function scenario uses a combination of operations on implicit set representations (support functions) and overapproximation steps.

Two operators are necessary to compute the reachable states: computing the states reachable by time elapse and computing the image of a set of states that take a transition.

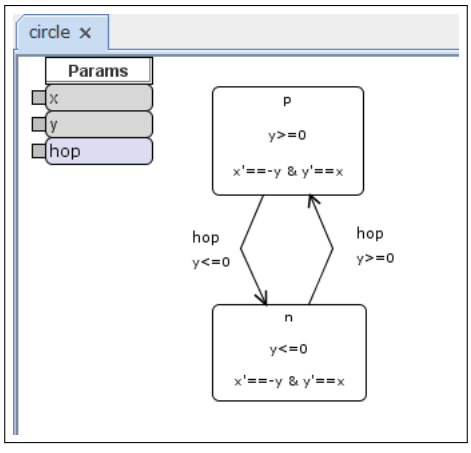
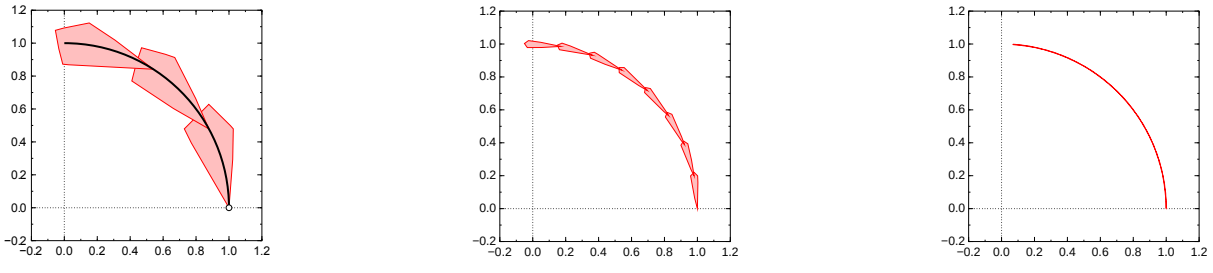


Figure 5.2: A two-dimensional system moving in circles around the origin



(a) $\delta = 0.5$ (b) $\delta = 0.2$ (c) $\delta = 0.05$

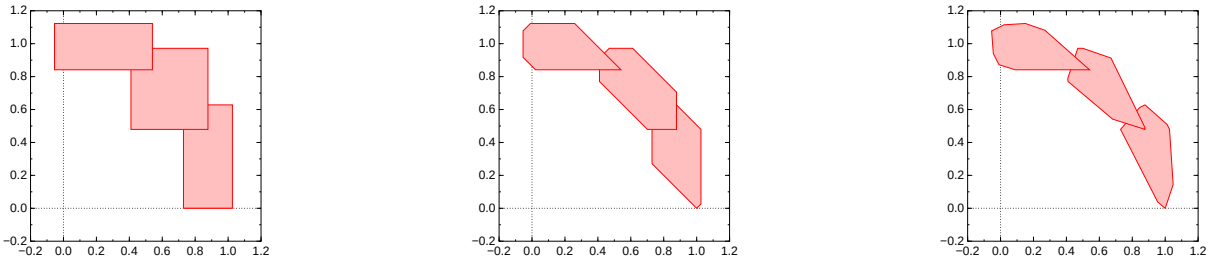
Figure 5.3: A flowpipe (bold in black) and the convex sets generated by SpaceEx to overapproximate it, for different values of the sampling time δ

In the following we consider what happens to a convex set of states in a single location if we let time elapse. Starting from the initial set, the LGG scenario computes a series of convex sets that cover the flowpipe. Each convex set covers a chunk of δ time out of the flowpipe, so that after having computed k of these sets we have covered the states that are reachable from X_0 up to time $k\delta$. We call δ the *sampling time*. Since we can't compute sets up to infinity, we define an upper bound on the time span we consider for each flowpipe, called the *local time horizon*. To know in detail about the construction of convex sets that cover the flowpipe, refer [GG09], [FLGD+11] and section 2.3.1.

Example 5.1. *Consider the system shown in Fig. 5.2. We consider location p, with dynamics*

$$\begin{aligned} \dot{x} &= -y, \\ \dot{y} &= x, \end{aligned}$$

which makes its states move around the origin in circular trajectories. We consider as initial set the state $(x = 1, y = 0)$, which gives rise to the circular flowpipe shown in bold in Fig. 5.3(a). The LGG time elapse algorithm with sampling time $\delta = 0.5$ and local time horizon 1.5 produces the three convex sets shown in Fig. 5.3(a), which cover the flowpipe. For smaller sampling times, the accuracy increases, as shown in Fig. 5.3(b) and Fig. 5.3(c).



(a) box (b) octagonal (c) 16 uniform

Figure 5.4: Flowpipe overapproximation for different choices of template directions

From looking at the example, it seems that by reducing the sampling time, we might get arbitrarily close to approximating the flowpipe. But there is another source of overapproximation: The final result of the LGG algorithm are template polyhedra, i.e., polyhedra whose faces have a direction that is given a priori. The use of template polyhedra allows one to avoid costly operations on polyhedra such as convex hull and existential quantification, which can be exponential in the number of variables. The price for this scalability is the degree of overapproximation that such an a-priori choice incurs. As the number of provided directions goes to infinity (assuming they are evenly distributed), the error goes to zero. In the worst case, the number of directions needed to fall under a given error bound is exponential in the number of variables. Experiments have shown that in practice a low number of directions may suffice, but this depends on the system and the property at hand.

The LGG scenario provides three options to choose the template directions for an n-dimensional system:

- *box* directions, i.e., $2n$ directions aligned with the axes, i.e., $x_i = \pm 1$, $x_k = 0$ for $k \neq i$;
- *octagonal* directions, i.e., $2n^2$ directions, consisting of all combinations of $x_i = \pm 1$, $x_j = \pm 1$, $x_k = 0$ for $k \neq i, j$;
- *uniform* directions, i.e., a set of m directions that are (as well as possible) uniformly distributed.

Example 5.2. *Figure 5.4 shows the flowpipe of the initial state* $(x = 1, y = 0)$ *with sampling time* $\delta = 0.5$ *and local time horizon* 1.5 *for box, octagonal and 16 uniformly distributed directions.*

Intersection with the invariant All states that are reachable within a location must satisfy the location's invariant. For the flowpipe computation, this is achieved by intersecting the polyhedra that cover the flowpipe with the invariant.

5.3.2.1 Computing successors of transitions

Each flowpipe that is created by the time elapse step is passed separately to the computation of transition successors. To compute the successor states we compute the states

that satisfy the guard, and then map them according to the assignment of the transition. The states that satisfy the invariant of the target location are the successor states.

Assignment If the assignment is invertible and deterministic, i.e., of the form $x := Ax + b_0$ with A being an invertible square matrix, the mapped states are computed exactly by mapping the polyhedron. Otherwise (non-invertible A or nondeterministic inputs), the mapped states are computing using a template overapproximation with the same template direction as used for time elapse.

Clustering Each flowpipe consists of a possibly large number of convex sets that cover the actual trajectories. When computing the states that can take a transition, clustering reduces this number. It iteratively replaces a group of sets of the flowpipe with a single convex set, their template hull. An option called *clustering percentage* determines how many sets come out of this process: A percentage of 0 means no reduction in the number of sets, all sets are passed to the aggregation step outlined below. A percentage of 100 means that all sets are combined into a single set (no aggregation necessary). A value between 0 and 100 groups the convex sets such that the relative distance (Hausdorff) to the original is below the given value (smaller values indicate higher accuracy). The sets coming out of the clustering then go through the aggregation step.

Aggregation The clustering step creates a certain number of convex sets, each one spawning its own flowpipe in the next time elapse computation. This may multiply the number of sets with each iteration, leading to an explosion in the number of sets and slowing the analysis to a halt. To avoid this effect and speed up the analysis, these sets can optionally be overapproximated by their convex hull. A faster but more coarse alternative is to set the clustering percentage to 100, which results in only one convex set (the template hull).

We describe how to use the flowpipe guard intersection with the LBS flowpipe guard intersection algorithm discussed in chapter 4 with the support function scenario.

5.3.2.2 Support Function Scenario with LBS intersection

The LBS intersection is activated by passing a positive argument to the **intersection-error** option. By default, LBS intersection algorithm is switched off and the standard intersection method is activated. The argument ϵ to the intersection-error option instructs LBS to keep searching for the support function of the flowpipe guard intersection set until the difference between the upper and the lower bound on the support function value is less than or equal to ϵ. Hence, specifying an argument 0 should return the most precise result. Recall that LBS computes the support function by solving a minimization problem and does the minima bracketing before running the sandwich algorithm [refer chapter 3]. After the minima bracketing, LBS finds a lower and upper bound to the minima. With the intersection-error switched on, the support function is computed up to atleast the difference of lower and upper bound of the support function as obtained after the minima bracketing and an epsilon specified larger than this will have no effect.

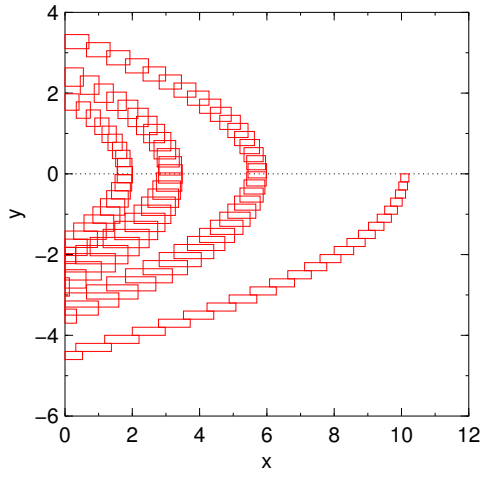
Figure 5.5 illustrates the effect of the intersection-error on the bouncing ball model shown in 4.6. Figure 5.5(a) is the most precise reachable set computed by SpaceEx with LBS

intersection method and Figure 5.5(c) is with the intersection-error set to 1. This is with box directions, sampling time as 0.2 and for four discrete jumps.

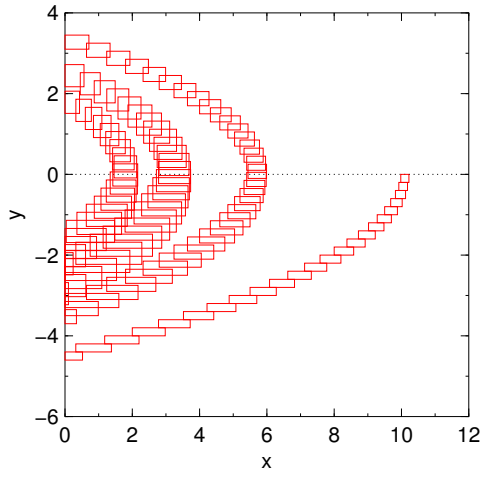
The LBS intersection method has a parameter called **minbrak**. This is to set the type of minima bracketing algorithm to use. Currently, there are two minima bracketing algorithms to choose from, which can be used with the LBS method - (1) Golden descent method and (2) Parabolic Extrapolation. Golden descent method is chosen by specifying the string "gold_desc" and the parabolic extrapolation method is chosen by specifying the string "parab_desc" to the minbrak option. If nothing is specified, golden descent method is chosen by default. Refer section 3.2.1 to recall the minima bracketing algorithms in detail.

There is another option named **intersection-method** to select the convex hull clustering with the LBS implementation. By default, the branch and bound algorithm to compute the convex hull of the intersection of the flowpipe with the guard set is used which can be explicitly set by specifying the string "lb_chull". The algorithm is explained in section 4.3.1. The simultaneous solve variant of LBS which does no clustering can be called with the string "lb_simult". Refer 4.3 for an explanation of this method.

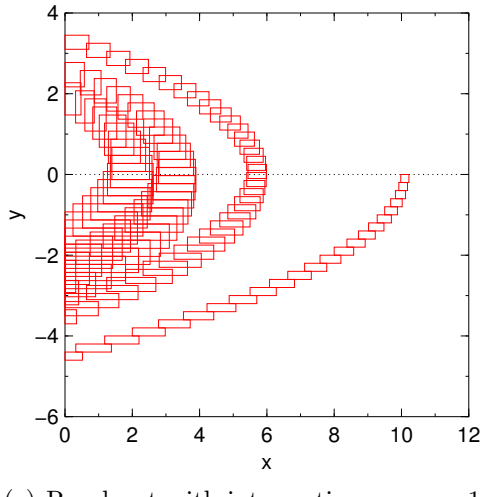
Lastly, there is an option named **split** which is used to define the split size. This number defines the size of the flowpipe which will be convex hull-ed before computing its intersection with the guard set. Split is set to 0 by default which means that all the intersecting flowpipe sections are convex hulled before computing the intersection with the guard set. Section 4.3.2 gives a detailed description of computing the flowpipe guard intersection with splitting.



(a) The most precise computation of the discrete jump with intersection-error as 0



(b) Reach set with intersection-error as 0.3



(c) Reach set with intersection-error as 1

Figure 5.5: Reachable Set computed by SpaceEx for three jumps in the bouncing ball model with different values of intersection-error parameter.

5.4 Software Engineering Behind SpaceEx

SpaceEx has been designed keeping in mind Robustness and Extendability as the goals. It is a tool platform which is more than being only a tool because it promises that new algorithms for hybrid systems reachability could be easily tested and integrated within its framework. Hence, Its design has been given special attention. In the next subsections, we discuss about the class structure design and design patterns used in SpaceEx, about smart pointers, version control, testing and debugging in SpaceEx.

5.4.1 Class Structure Design

In section 5.2, we discussed about what we think are the principle elements of a reachability analysis tool and the different possibilities of concretizing them. For example, the continuous post operator is a principle element and this may have different implementations depending on the type of dynamics in the hybrid automata and the type of continuous set representation. Also, the type of continuous set representation might depend on the dynamics of the system and the reachability algorithm per se. Thus we see that there is a interdependency. The goal is to have a class design which preserves data encapsulation and also provides ample scope for reusability and extendability. For the purpose of extendability, we defined the principle elements as abstract base classes. Different possible concretizations of the principle elements are then implemented as the derived classes. To counter for the fact that there is a compatibility issue on the type of concretizations of the principle elements that goes together and at the same time there is a need of data encapsulation, we implement a separate scenario class which integrates the different concretizations and take care of the compatibility issue. Thus for example, a polyhedron class derived from the continuous set abstract class is implemented without considering the type of dynamics or the reachability algorithm. A scenario class acts as a placeholder for each of the principle elements and chooses the right combination of their concretizations. This type of class design conforms with the **Abstract Factory** design pattern. The different design patterns that we mention in this section are explained in the book [GHJV94].

Figure 5.6 shows the class hierarchy diagram of the *post operator* in SpaceEx as an example. The *post_operator* class shown in the diagram is a abstract base class and shown are the concretizations.

As a tool platform and as an experimental framework, SpaceEx defines a family of algorithms for reachability, operations on sets, input/output etc. It conforms to the **Strategy design pattern** as much as possible which lets the algorithm vary independently from clients that use it. Different algorithms to perform a task are encapsulated as different subclass implementations of the common abstract base class. For example, in the support function scenario implementation in SpaceEx, we compute the support function of polytopes which are nothing but LP problems. There are different algorithms to solve a LP problem which are encapsulated as separate classes to a abstract base class called *lp_solver*. Figure 5.7 shows the class hierarchy diagram of LP solvers in SpaceEx. The subclasses *lp_solver_fm*, *lp_solver_glpk* and *lp_solver_monniaux* implements different algorithms to solve a LP problem.

SpaceEx reachability algorithm implementation often works on aggregate objects like a

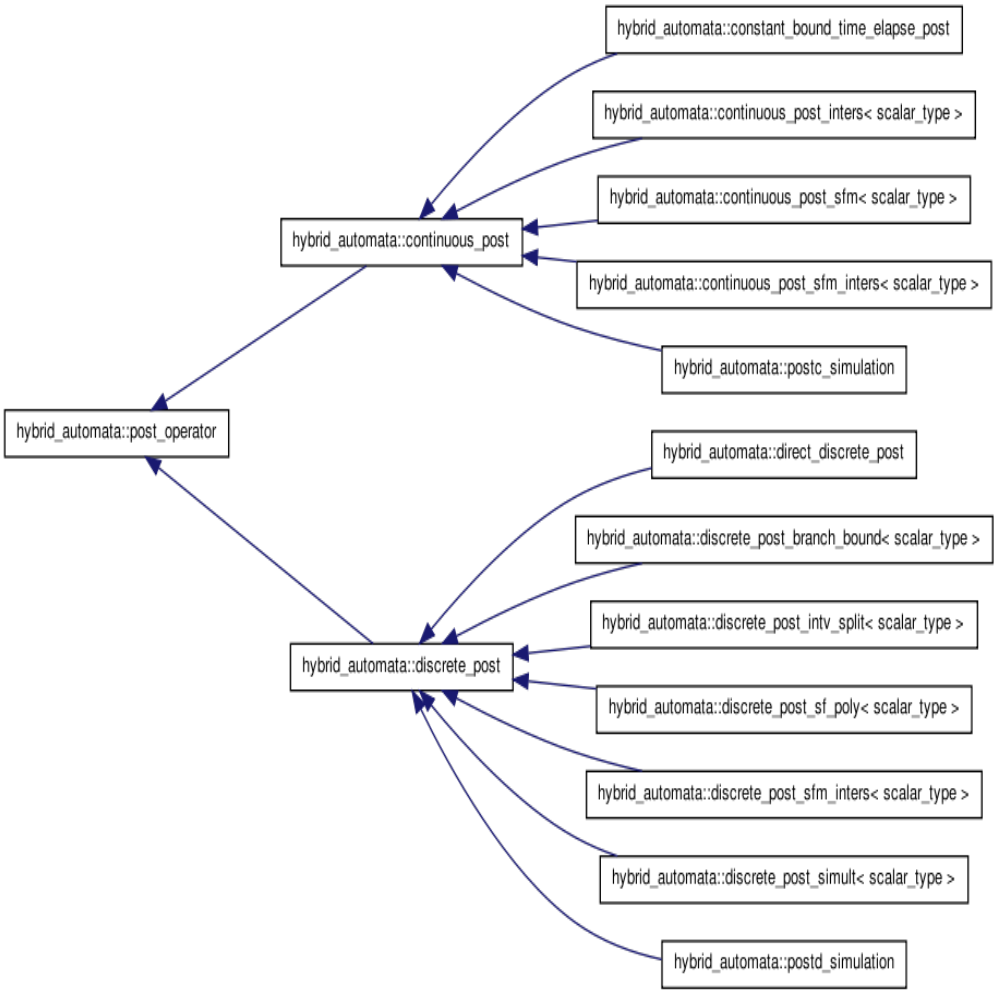


Figure 5.6: Class hierarchy diagram of the post operator in SpaceEx.

collection of continuous sets, a collection of symbolic sets etc. The class design therefore must provide a way to access the elements of the aggregate objects without exposing the underlying representation. The **Iterator design pattern** is used wherein the aggregate class have methods which creates and returns an iterator object specific to its own type.

Visitor design pattern has been used where we could identify a number of different operations on a class of objects and the operations depend on the concrete type of the operand object. Figure 5.8 shows the hybrid automaton visitor class hierarchy diagram. CIF_automaton_formatter, SX_automaton_formatter, print, automaton_to_supp_f_adapter etc defines the different operations on the hybrid automaton.

To give an idea of the size of SpaceEx, It consists of 30 namespaces excluding the std namespace, 628 classes, a total of 571 files (C++ header and source files), 65683 lines of C++ source code (without comments and blank lines) and a total of 20543 lines of comments to date.

A HTML documentation of SpaceEx which shows its complete interface with class hierarchy, collaboration diagram and dependency graphs can be generated from the source code using the automated document generation tool called doxygen [fSCD].

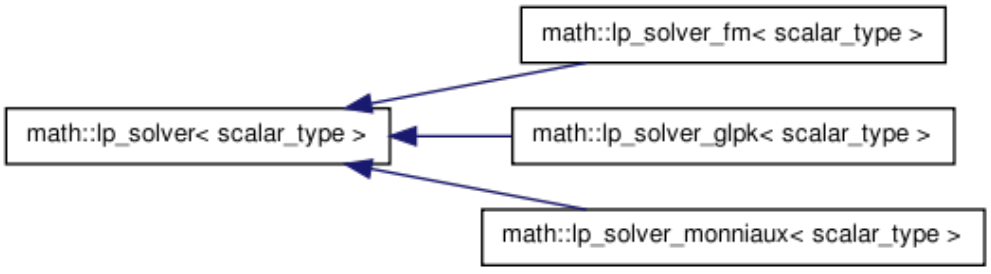


Figure 5.7: Class hierarchy diagram of LP solvers conforming to the Strategy Design Pattern

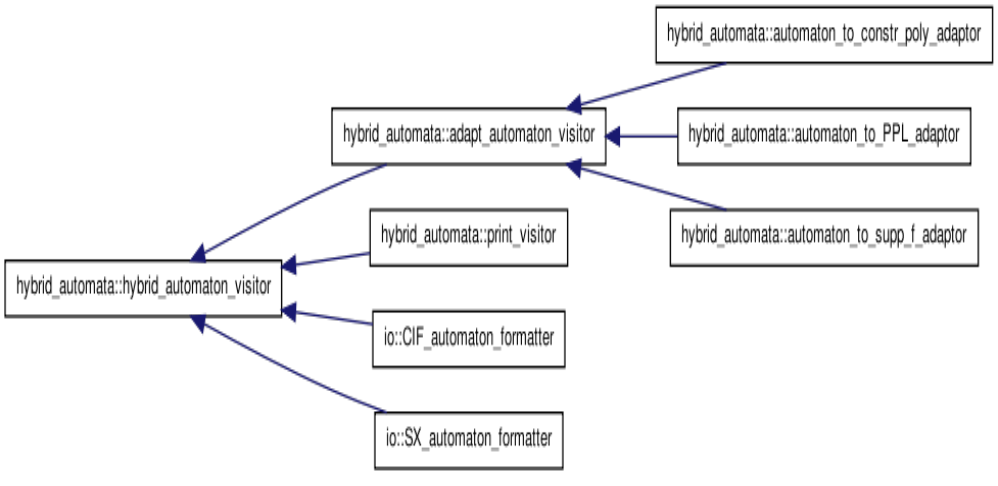


Figure 5.8: Class hierarchy diagram of the hybrid automaton visitor

5.4.2 Smart Pointers

A smart pointer is a C++ class that mimics a regular pointer in syntax and some semantics, but in addition provides value semantics. An object with value semantics is an object that one can *copy* and *assign to*. Compilers do not take care of the memory management for normal pointers. For example, when a pointer is assigned a value with the new operator, it becomes the owner of the object it points to. This pointer has to be explicitly deleted with the delete operator for this memory to be released. In the case of copying normal pointers, both the copied and the original pointers then own the object it points to. Consequently, if one of them is deleted, the memory allocated to the object is released. Moreover, double deletion could be catastrophic. Therefore, normal pointers do not have value semantics. Smart pointers does all the required memory management for the user and hence are called 'smart'. SpaceEx is implemented with the the boost smart pointers which is called **shared_ptr**. A documentation of the boost shared pointer can be found at [oBSP].

The reader is referred to [Ale01] for a deeper insight into the implementation of smart pointers.

5.4.3 Revision Control

Revision control is the process of managing multiple versions of a piece of information. One could list down a number of reasons as to why have multiple versions.

- In the context of software development, having multiple versions stored serves as a development timeline and may help to reflect back and learn from its own evolution.
- Another reason could be pointed to the changing nature of the world. Requirements of a software keep changing over time - the algorithms implemented in a version may stand old and less efficient compared to the newer versions or sometimes the old versions may be found better suited later in the future. Hence, it is always a good idea to keep the older versions in the development process which would allow to revert back when needed.
- Version control also helps in recovering from errors. If a new change turns out to be faulty, then one can revert back to an older working version. Revision control if used wisely could largely help in the debugging process as well.
- Revision control helps in the collaborative development process. For most of the cases, softwares are developed by a number of developers, many a times distributed geographically. Developers may write code which conflict with each other and needs to be resolved. A good revision control system must be able to resolve such conflicts.

Revision control manually is tiresome and error prone for even small scale software development projects. There are a number of automated revision control tools like CVS [Sysa], Subversion [Sysd], GIT [Sysb] and Mercurial [Sysc]. The main difference between these are some of them like CVS and Subversion have a centralized server/client architecture whereas others like GIT and Mercurial have a distributed architecture. In the centralized tools, the repository is stored in a single server and clients communicate with the repository over the network. In the distributed tools, there can be multiple local repositories cloned from a repository for each client or user.

Distributed version control is relatively newer and has the following advantages over the centralized counterpart.

- A user can communicate with its own copy of the repository without the need of having a network.
- Distributed version control systems are faster because most metadata is stored locally unlike in the centralized tools where most metadata is stored in the central server which needs to be updated over the network.
- Distributed version control systems are robust. If the repository in the centralized version control software gets corrupted then the repository is lost unless there is a backup. In the distributed counterpart, there are multiple copies of the repository distributed among the users. All getting corrupted at the same time is rare.

Mercurial is a distributed version control software. SpaceEx version control is done with Mercurial because of the above mentioned advantages of distributed architecture. The

distributed architecture proves to be helpful in the development of different features in different independent feature branches. Feature branches is a good way of managing changes in large projects by breaking up and defining independent branches. A group of developers has a shared branch of its own, cloned from a single master branch [O'S09]. Developers can work on a particular branch independently and isolated from other branches. When a particular feature is in a good shape, someone on that branch pulls and merges the master branch into the feature branch then pushes back up to the master branch. There could be an additional level of supervision in the master branch as to what new features got to be added and what all needs to be discarded. The supervisor in the master branch could pull in only the changes it thinks that should go in from the pushed feature branch and discard the rest.

The idea of feature branches can also be implemented without having clones for each branch in mercurial. Mercurial treats all of the development history as a series of branches and merges. Feature branches can be implemented just by giving a persistent *name* to a branch. By default, all commits goes to the *default* branch in mercurial. New branches can be created with the command:

```
$ hg branch new_branch
```

Switching between branches is done with the *update* command:

```
$ hg update main_branch
```

SpaceEx development utilizes this feature branches approach with mercurial in the development of its different features with branch naming as explained above and not with repository cloning for each branch. Figure Figure 5.4.3 illustrates feature branching in SpaceEx.

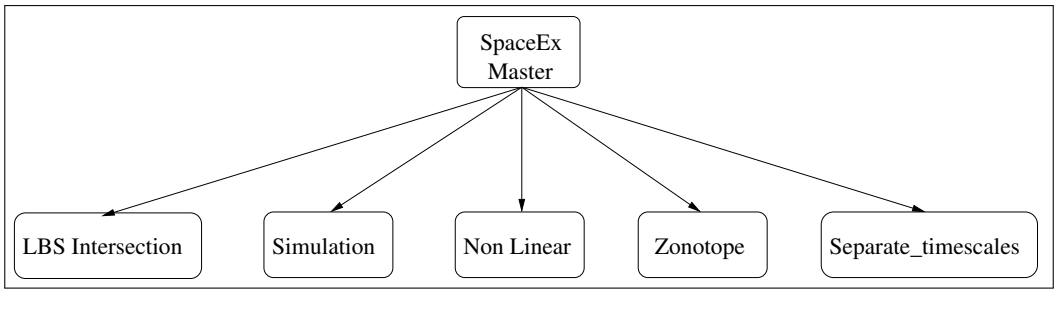


Figure 5.9: Illustrating feature branch development in SpaceEx

Mercurial is a free software and it is open source. It is easy to use and most of its commands are the same with the classical rcs tools like SVN and CVS. Mercurial is portable to all popular operating systems. [O'S09] provides a comprehensive guide to using Mercurial for revision control in software development.

5.4.4 Testing and Debugging

Testing the individual software units during the development process is known as unit testing. Unit testing is a better practice than testing software modules consisting of a bunch of software units. The later increases the complexity of testing because of the fact that there are more candidates where the identified bug(s) could exist and because of the possible interdependence between the different components under test, the exact cause of the bug might be hard to find. In short, unit testing finds the bugs early in the development cycle.

Unit testing gives greater flexibility in the development process. Since the correctness of the individual software units is tested, refactoring becomes simpler. Refactoring means disciplined restructuring of the existing parts of the software without changing the overall desired behavior.

In the continuous test framework, the test cases for the units persist and they are run as the software undergoes change. If any change causes a failure in one or more of the unit tests, that signals an unintended effect of the change over the software units whose tests fail. Hence, continuous unit testing gives greater confidence in the integrity of the software as it grows.

SpaceEx has undertaken a test-driven development using the UnitTest++ package [Pac] for unit testing. UnitTest++ is a lightweight unit testing framework in C++. UnitTest++ provides with a number of macros. There are basically three types of macros, TEST Macro, SUITE macro and CHECK macros. A TEST macro is the basis of a test. A SUITE macro is a group of TEST macros and provides them with a namespace. CHECK macros perform comparisons and outputs true or false results. A false result means that the test in which the check occurred has failed. UnitTest++ is easy to use and could be used for a first experience of testing in the software development process. A total of 503 unit tests in 121 test files were written in the development of SpaceEx so far. The testers consists of 14272 lines of code and 4149 lines of comments. The total testing time with UnitTest++ for all the testers is about 193.876 seconds on a standard x86 machine with 32 bits operating system.

GDB (The GNU Project Debugger) [Deb] has been used to debug the test failures and the runtime exceptions in SpaceEx. GDB is a commonly used debugger distributed with almost all Unix distributions. With GDB, breakpoint can be set at desired points in a program from where the program execution could be traced line by line. Program variables could be monitored at each step. GDB can also display the call stack when there is a crash.

5.5 Models in SpaceEx

A SpaceEx model can be created using the SpaceEx Model editor which stores the model in the *SX* format. A model is made up of one or several components. There are two types of components: a *base* component which corresponds to a single hybrid automata. A *network* component consists of one or more instantiations of other components (base or network) and corresponds to a set of hybrid automata in parallel composition. Refer [Fre] for a detailed description of a SpaceEx model.

5.6 Libraries

SpaceEx has its own rich library for most of the math operation and data structures it requires. In addition, the analysis core of SpaceEx uses a number of third party libraries. The PHAVer scenario uses the PPL library [BRZH02] to represent continuous sets as polyhedra. The Boost C++ library [Liba] is used for its fast and efficient data structures in C++. The support function scenario uses the GLPK (GNU Linear Programming Kit) library [Mak09] to compute the support function of polyhedra which are nothing but LP problems. SpaceEx uses GMP - the GNU Multiple Precision arithmetic library [Libb] for arbitrary precision arithmetic. Table 5.1 lists the third party libraries that SpaceEx core uses, to whose authors we are most grateful.

SpaceEx is released under the GNU GPL version 3 license [vL] which is compatible with the licenses of the third party libraries that it uses [Table 5.2].

Table 5.1: Third party libraries used in SpaceEx.

Name	Version	Year	Author(s)
Parma Polyhedra Library	0.11	2011	R. Bagnara, P. M. Hill, E. Zaffanella
Boost C++ Libraries	1.46.1	2011	multiple
GNU Multiple Precision Arithmetic Library	5.0.2	2011	multiple
GNU Linear Programming Kit	4.45	2010	multiple
SUNDIALS (Solver Suite)	2.4.0	2009	R. Serban, C. Woodward, A. Hindmarsh
ublasJama	1.0.2.2	2005	Frederic Devernay
TinyXML	2.5.3	2007	Lee Thomason

Table 5.2: Licenses of the third party libraries used in SpaceEx.

Name	License
Parma Polyhedra Library	GNU GPLv3
Boost C++ Libraries	Boost Software License
GNU Multiple Precision Arithmetic Library	GNU LGPL
GNU Linear Programming Kit	GNU GPLv3
SUNDIALS (Solver Suite)	BSD License
ublasJama	Boost Software License
TinyXML	zlib License

5.7 SpaceEx Output

SpaceEx provides with four output formats - (1) Textual (**TXT**) (2) Vertice List (**GEN**) (3) 3D Visualization (**JVX**) and (4) [min,max] interval on the output variables (**INTV**). [Fre] gives a detailed documentation of output formats (1),(2) and (3). In addition, the **INTV** format which stands for interval format has been later added to the cavalry. It outputs the minimum, maximum interval on the continuous variables of the system globally as well as location-wise for the given analysis configuration. The output variables could be chosen with the -a option in SpaceEx.

Chapter 6

Conclusion and Future Work

This chapter briefly summarizes the contributions of this thesis and suggest some possible directions for future work.

The contributions of this thesis are as follows:

1. **The SpaceEx tool platform for safety verification of hybrid systems**: This thesis provides an extendable tool platform, *SpaceEx* on which algorithms for safety verification of hybrid systems can be implemented. We put an effort in identifying the principal elements of such a tool platform and provide the generic implementation of those principal elements. SpaceEx tool platform provides placeholders for the elements which may vary from approach to approach, e.g., the continuous set representation. Hence, such an extendable tool platform should aid the researchers in experimenting with their novel methods to reachability analysis for safety verification, CEGAR based algorithms etc. The availability of the already implemented principal elements should save a lot of development time of users.

2. As a practical demonstration of the tool platform, two different scenarios have been implemented. The **PHAVer scenario** and the **Support Function** scenario. The PHAVer scenario is for the safety verification of linear hybrid automata (LHA) with polyhedra as the continuous set representation. The Support Function scenario is for safety verification of hybrid automata with affine dynamics and affine maps over the discrete transitions. The Support Function scenario implements the support function based reachability algorithm proposed in [GG09] where continuous sets are represented as convex sets defined by their support functions. Both implementations are real usable tool implementations and not just prototypes. This demonstrates the usefulness of the SpaceEx tool platform.

3. Large over-approximation while computing the transition successors limits the use of the Support Function approach for the analysis of hybrid systems with frequent discrete jumps. A more precise support function based algorithm for transition successor computation is proposed in this thesis. The proposed method largely improves the accuracy of the discrete image computation during transitions in hybrid systems. The accuracy improvements are shown with some case studies. The scalability is illustrated on the filtered oscillator case study with up to 130 variables. This precise discrete image computation has been implemented on the support function scenario of the SpaceEx tool platform.

4. As part of the precise discrete image computation, a new algorithm for the minimization of univariate convex function is proposed which has been named **Lower Bound**

Search (LBS) algorithm.

Some directions for future work:

• We mentioned in section 2.3.2 that we have a support function representation of the flowpipe using the reachability algorithm proposed in [GG09]. We compute an outer polyhedral approximation of this set by sampling the support function in the template directions, which are fixed a priori. This conversion from support function to polyhedra representation is used for operations like intersection and containment which can be cheaply carried out on constraint represented polyhedra. An arbitrarily chosen set of template directions for the outer-polyhedral computation from support function may result in high over-approximation error. Sampling the support function in a lot of directions will result in less approximation error but we have to pay in computation time. Synthesizing a set of well chosen template directions will provide a balance between the approximation error and the computation time. How to synthesize such a set of directions remains a future direction of research. In our lower bound search algorithm to compute the support function of the flowpipe-guard intersection in a given direction [alg. 3.2.2.4], we compute the support function of the flowpipe in a number of directions (each parameter value λ corresponds to a direction) during the minima search of the support function, which is a convex function. The algorithm stops when we find a direction which minimizes the function. It will be interesting to see the effect of adding this minimizing direction to the set of template directions for further computation of the flowpipe. Considering the vector field of the location dynamics may also provide us helpful clues to synthesize template directions.

• Section 4.3 illustrates the simultaneous solution of the multiple minimization problem for flowpipe-guard intersection with the proposed LBS algorithm. For computational speed up, it will be interesting to parallelize each such minimization problem. The implementation could be carried out in a multicore architecture. Similarly, the simultaneous execution of the lower bound search algorithm with branch and bound illustrated in section 4.3.1 can also be parallelized.

• We defined Support Function Matrix (SFM) as a data structure for storing and manipulating flowpipe representations in section 2.3.2. It will be interesting to implement the support function based reachability algorithm with SFMs in a **GPU** (Graphical Processing Unit) architecture since GPUs are highly efficient for matrix and vector operations. Though GPUs are specialized for graphical computations, application with high use of matrix and vector operations can also benefit the processing power of GPUs.

Bibliography

[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(1):3–34, 1995. 5, 8

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994. 8

[ADG03] Eugene Asarin, Thao Dang, and Antoine Girard. Reachability analysis of nonlinear systems using conservative approximation. In *HSCC'03*, volume 2623 in LNCS, pages 20–35. Springer, 2003. 82

[ADM02] Eugene Asarin, Thao Dang, and Oded Maler. The d/dt tool for verification of hybrid systems. In *CAV*, pages 365–370, 2002. 82

[Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison Wesley, 13 february, 2001. 95

[BBB07] Alberto Bemporad, Antonio Bicchi, and Giorgio C. Buttazzo, editors. *Hybrid Systems: Computation and Control, 10th International Workshop, HSCC 2007, Pisa, Italy, April 3-5, 2007, Proceedings*, volume 4416 of *lncs*. Springer, 2007. 106

[BDM+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 1998. 5

[BHR91] R. E. Burkard, H. W. Hamacher, and G. Rote. Sandwich approximation of univariate convex functions with an application to separable convex programming. *Naval Res. Logistics*, 38:911–924, 1991. 37, 43, 44

[BHZ08] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(12):3 – 21, 2008. ¡ce:title¿Special Issue on Second issue of experimental software and toolkits (EST)¡/ce:title¿. 15

[BLL+96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal - a tool suite for automatic verification of real-time systems, 1996. 5, 12

[BRZH02] Roberto Bagnara, Elisa Ricci, Enea Zaffanella, and Patricia M. Hill. Possibly not closed convex polyhedra and the parma polyhedra library. In Manuel V. Hermenegildo and Germán Puebla, editors, *SAS*, volume 2477 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2002. 99

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000. 8

[CK98] Alongkrit Chutinan and Bruce H. Krogh. Computing polyhedral approximations to flow pipes for dynamic systems. In *In Proceedings of the 37rd IEEE Conference on Decision and Control*. IEEE Press, 1998. 21, 23

[DBLY02] Alexandre David, Gerd Behrmann, Kim Guldstrand Larsen, and Wang Yi. A tool architecture for the next generation of uppaal. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th Anniv. Colloq. UNU/IIST*, volume 2757 of *lncs*, pages 352–366. Springer, 2002. 12

[Deb] The GNU Project Debugger. http://www.gnu.org/software/gdb/. 98

[Dis] Space Shuttle Columbia Disaster. http://en.wikipedia.org/wiki/Space_Shuttle_Columbia_disaster. 3

[DT97] George B. Dantzig and Mukund N. Thapa. *Linear Programming 1: Introduction*. Springer, 1997. 15

[DT03] George B. Dantzig and Mukund N. Thapa. *Linear Programming 2: Theory and Extensions*. Springer, 2003. 15, 30

[FI04] Ansgar Fehnker and Franjo Ivancic. Benchmarks for hybrid systems verification. In Rajeev Alur and George J. Pappas, editors, *HSCC*, volume 2993 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2004. 72, 80

[FJK08] Goran Frehse, Sumit Kumar Jha, and Bruce H. Krogh. A counterexample-guided approach to parameter synthesis for linear hybrid automata. In Magnus Egerstedt and Bud Mishra, editors, *HSCC*, volume 4981 of *lncs*, pages 187–200. Springer, 2008. 82

[FKR06] Goran Frehse, Bruce H. Krogh, and Rob A. Rutenbar. Verifying analog oscillator circuits using forward/backward abstraction refinement. In Georges G. E. Gielen, editor, *DATE*, pages 257–262, 2006. 82

[FLGD+11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In Shaz Qadeer Ganesh Gopalakrishnan, editor, *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, LNCS. Springer, 2011. 11, 12, 26, 27, 29, 81, 88

[FR09] Goran Frehse and Rajarshi Ray. Design principles for an extendable verification tool for hybrid systems. In *Proceedings of ADHS'09*, volume 3, part 1, 2009. 11, 64, 81

[Fre] Goran Frehse. An introduction to spaceex v0.8. http://spaceex.imag.fr/documentation/user-documentation/introduction-spaceex-27. 81, 98, 99

[Fre08] Goran Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. *STTT*, 10(3):263–279, 2008. 5, 12, 82, 86

[fSCD] Document Generator from Source Code (Doxygen). http://www.stack.nl/~dimitri/doxygen/. 94

[Fuk99] Komei Fukuda. cdd/cdd+ reference manual, 1999. 15

[GG09] Colas Le Guernic and Antoine Girard. Reachability analysis of hybrid systems using support functions. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 540–554. Springer, 2009. 9, 23, 33, 60, 88, 101, 102

[GGM06] Antoine Girard, Colas Le Guernic, and Oded Maler. Efficient computation of reachable sets of linear time-invariant systems with inputs. In João P. Hespanha and Ashish Tiwari, editors, *HSCC*, volume 3927 of *lncs*, pages 257–271. Springer, 2006. 82

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition (November 10, 1994). 93

[Gir04] Antoine Girard. *Analyse Algorithmique des Systemes Hybrides*. PhD thesis, Institut National Polytecnique de Grenoble, 2004. 22

[Gir05] Antoine Girard. Reachability of uncertain linear systems using zonotopes. In Manfred Morari and Lothar Thiele, editors, *HSCC*, volume 3414 of *Lecture Notes in Computer Science*, pages 291–305. Springer, 2005. 23, 82

[GJ01] Ewgenij Gawrilow and Michael Joswig. Polymake: an approach to modular software design in computational geometry. In *Symposium on Computational Geometry*, pages 222–231, 2001. 15

[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. 16, 18

[GL08] Antoine Girard and Colas Le Guernic. Efficient reachability analysis for linear systems using support functions. In *Proc. IFAC World Congress*, 2008. 64, 82, 86, 87

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *IEEE Symp. Logic in Computer Science*, page 278, Washington, DC, USA, 1996. IEEE Computer Society. 5

[HHWT97] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110 – 122, 1997. 5, 82

[HKPV95] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? In *Journal of Computer and System Sciences*, pages 373–382. ACM Press, 1995. 8

[HNSY92] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:394–406, 1992. 11

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997. 5

[JGP99] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 7, 1999. 4

[LD60] A. H. Land and A. G Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960. 68

[LG09] Colas Le Guernic. *Reachability analysis of hybrid systems with linear continuous dynamics*. PhD thesis, Université Grenoble 1 - Joseph Fourier, 2009. 33, 56

[Liba] Boost C++ Libraries. http://www.boost.org/. 99

[Libb] GNU Multiple Precision Arithmetic Library. http://gmplib.org/. 99

[Lio96] J.L. Lions. Ariane 5 flight 501 software failure. *Report of the inquiry board*, July 1996. Available at http://www.esa.int. 3

[Liv08] M. Livio. *The Golden Ratio: The Story of Phi, the World's Most Astonishing Number*. Paw Prints, 2008. 39

[Mak09] Andrew Makhorin. GNU Linear Programming Kit, v.4.37, 2009. http://www.gnu.org/software/glpk. 87, 99

[Mit07] Ian M. Mitchell. Comparing forward and backward reachability as tools for safety analysis. In Bemporad et al. [BBB07], pages 428–443. 83

[oBSP] Documentation of Boost Smart Pointer. http://www.boost.org/doc/libs/1_48_0/libs/smart_ptr/shared_ptr.htm. 95

[oPFP] A Library of Polyhedral Functions (Polylib). http://www.irisa.fr/polylib/. 15

[O'S09] Bryan O'Sullivan. *Mercurial: The definitive Guide*. 0'Reilly, 2009. 97

[Pac] C++ Unit Testing Package. http://unittest-cpp.sourceforge.net/. 98

[PVTF02] William H. Press, William T. Vetterling, Saul A. Teukolsky, and Brian P. Flannery. *Numerical Recipes in C++: the art of scientific computing*. Cambridge University Press, New York, NY, USA, 2nd edition, 2002. 37, 40

[Roc70] R. Tyrrell Rockafellar. *Convex Analysis*. Princeton Univ Pr, (June 1970). 14, 36

[RR92] Gnter Rote and G Unter Rote. The convergence rate of the sandwich algorithm for approximating convex functions. *Computing*, 48:337–361, 1992. 36, 43, 44

[Sch93] Rolf Schneider. *Convex bodies : The Brunn-Minkowski Theory*. Cambridge University Press, February 26, 1993. 14, 33

[SDI08] Sriram Sankaranarayanan, Thao Dang, and Franjo Ivancic. Symbolic model checking of hybrid systems using template polyhedra. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2008. 17

[Sim] The Mathworks MATLAB Simulink. http://www.mathworks.in/products/simulink/. 7

[SSM05] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In Radhia Cousot, editor, *Proc. of Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3385 of *lncs*, pages 21–47, Paris, France, January 2005. Springer Verlag. 17

[Str86] Bjarne Stroustrup. *The C++ Programming Language, First Edition*. Addison-Wesley, 1986. 81

[Sysa] CVS-Version Control System. http://cvs.nongnu.org/. 96

[Sysb] GIT-Distributed Version Control System. http://git-scm.com/. 96

[Sysc] Mercurial Distributed Version Control System. http://mercurial.selenic.com/. 96

[Sysd] Subversion-Version Control System. http://subversion.apache.org/. 96

[Tiw08] Hans Raj Tiwary. On the hardness of computing intersection, union and minkowski sum of polytopes. *Discrete & Computational Geometry*, 40(3):469–479, 2008. 68

[vL] GNU GPL version 3 License. http://www.gnu.org/licenses/gpl.txt. 99

[Zie95] Gnter M. Ziegler. *Lectures on Polytopes*, volume 152 of Graduate Texts in Mathematics. Springer, 1995. 15