UNIVERSITY OF CALIFORNIA,
IRVINE

Formal Verification of AI-Controlled Cyber-Physical Systems Using Polynomial Approximations: Constraints Solver, Model Checkers, and Applications

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Wael Fatnassi

2024

DEDICATION

To my wife, for her love, patience, and unwavering support
To my family for their endless encouragement.
To my dog Leo, whose joyful presence brought laughter into my days.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I am profoundly thankful for the guidance, support, and trust bestowed upon me by my advisor, Prof. Yasser Shoukry. He accepted me into the Ph.D. program, believed in my potential, and provided invaluable advice across technical, professional, and personal realms. His mentorship has been a cornerstone of my journey, shaping me into the researcher and individual I am today.

My gratitude extends to my committee members, Prof. Aparna Chandramowlishwaran and Prof. Yanning Shen. Their insightful feedback and rigorous review of my work have significantly contributed to its depth and quality.

I am also indebted to my colleagues at the Resilient Cyber-Physical Systems Lab. Special thanks to Ulices Santa Cruz Leal, Haitham Khedr, Momina Sajid, James Ferlez, Kohei Tsujio, Mahmoud ElFar, Goli Vaisi, and Jaejeong Park for sharing this journey and for the camaraderie that has made this experience truly enriching.

Above all, my deepest gratitude is for my family, whose unconditional love and unwavering support have been my bedrock. Their faith in me through every high and low has been the light guiding me forward. I am because of them.

VITA

Wael Fatnassi

Ph.D. degree in Electrical and Computer Engineering	**2024**
University of California, Irvine	
Research and Teaching Assistant	**2019–2024**
University of California, Irvine	
Winter Internship	**Winter 2024**
HRL Laboratories	
Master of Science in Electrical Engineering	**2017–2019**
University of Idaho	
Graduate and Professional Student Association (GPSA) Award	**2018**
University of Idaho	
Summer Research Internship	**Summer 2018**
Idaho National Laboratory (INL)	
Bachelor of Science in Telecommunication Engineering	**2013–2016**
Higher School of Communications (Sup'Com)	
Engineering Internship	**Summer 2016**
HTWK Leipzig	
Deutscher Akademischer Austauschdienst (DAAD) Scholarship	**2016**
HTWK Leipzig	

ABSTRACT OF THE DISSERTATION

Formal Verification of AI-Controlled Cyber-Physical Systems Using Polynomial Approximations: Constraints Solver, Model Checkers, and Applications

By

Wael Fatnassi

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2024

Associate Professor Yasser Shoukry, Chair

The last decade's advancement in machine learning (ML) for controlling cyber-physical systems has heralded a new era in autonomous technology, driving innovations from self-driving cars to smart infrastructure. However, these systems often grapple with challenges related to safety, reliability, and the ability to generalize across different scenarios. This dissertation aims to bridge the gap between the scalability and flexibility of ML-based control systems and the rigorous safety and reliability guarantees provided by formal methods and control theory. It introduces novel methodologies that leverage machine learning to enhance the design, verification, and optimization of AI-controlled cyber-physical systems, ensuring they meet high-level specifications while managing such systems' inherent complexity and non-linearity.

The contributions of this thesis are multi-fold. 1) We proposed a highly efficient and parallelizable solver called PolyAR, which aims to solve general multivariate polynomial inequality constraints. PolyAR uses convex polynomials as an abstraction for highly nonlinear polynomials. Such abstractions were previously shown to be powerful to prune the search space and restrict the usage of sound and complete solvers to small search space. We compared the scalability of PolyAR against state-of-the-art solvers such as Z3 8.9 and Yices 2.6 on complex design and verification problems. The experiment results show that the PolyAR solver drastically outperformed Z3 8.9 and Yices 2.6 regarding execution time. 2) We developed PolyARBerNN, an enhancement to PolyAR that employs neural networks (NN) to guide the abstraction refinement procedure that helps to select the right abstraction out of a set of pre-defined abstractions and a Bernstein polynomial-based search space pruning mechanism. These enhancements together made PolyARBerNN capable of solving complex instances and scaling more favorably compared to the state-of-the-art nonlinear real arithmetic solvers while maintaining the soundness and completeness of the resulting solver. In addition, we proposed an efficient optimizer called PolyAROpt that transforms polynomial objective functions into polynomial constraints (on the gradient of the objective function) whose solutions are guaranteed to be close to the global optima. PolyAROpt optimizer uses PolyARBerNN to solve constrained polynomial optimization problems. Numerical results show that PolyAROpt can solve high-dimensional and high-order polynomial optimization problems faster than the built-in optimizer in the Z3 8.9 solver. 3) We proposed an efficient algorithm called BERN-NN that employs polynomial interval arithmetic, where tight over/under approximations of the NN's activation functions are computed using Bernstein polynomials. These polynomials have several interesting mathematical proprieties. One particular property is called the sharpness propriety, which allows us to obtain extremely tight bounds that are tighter than those currently exist in the literature (e.g., interval arithmetic, crowns, linear programming, and many centered forms). Moreover, we exploited the mathematical proprieties of Bernstein polynomials to convert the proposed polynomial interval arithmetic operations into add-and-multiply operations, which are easily implemented using GPUs. Thanks to those GPUs, our tool's execution time is drastically reduced. Experimental results show that our method approximates NN's outputs tighter than state-of-the-art NN verification tools by several orders of magnitude. 4) We proposed BERN-NN-IBF, a significant enhancement of the Bernstein-polynomial-based bound propagation algorithms. BERN-NN-IBF offers three main contributions: (i) a memory-efficient encoding of Bernstein-polynomials to scale the bound propagation algorithms, (ii) optimized tensor operations for the new polynomial encoding to maintain the integrity of the bounds while enhancing computational efficiency, and (iii) tighter under-approximations of the ReLU activation function using quadratic polynomials tailored to minimize approximation errors. Through comprehensive testing, we demonstrate that BERN-NN-IBF achieves tighter bounds and higher computational efficiency than the original BERN-NN and state-of-the-art methods, including linear programming and convex used within the winner of the VNN-COMPETITION.

Part I

Using Machine Learning to Design Novel Solvers For Analyzing Complex Cyber-Physical Systems

Chapter 1

PolyAR: A Highly Parallelizable Solver For Polynomial Inequality Constraints Using Convex Abstraction Refinement

This chapter presents PolyAR, a highly parallelizable solver for polynomial inequality constraints. PolyAR provides several key contributions. First, it uses convex relaxations of the problem to accelerate the process of finding a solution to the set of the non-convex multivariate polynomials. Second, it utilizes an iterative convex abstraction refinement process which aims to prune the search space and identify regions for which the convex relaxation fails to solve the problem. Third, it allows for a highly parallelizable usage of off-the-shelf solvers to analyze the regions in which the convex relaxation failed to provide solutions. We compared the scalability of PolyAR against Z3 8.9 and Yices 2.6 on control designing problems. Finally, we demonstrate the performance of PolyAR on designing switching signals for continuous-time linear switching systems.

1.1 Introduction

Advances in constraints programming have opened several venues for control system synthesis and verification of hybrid systems. For instance, linear programming and convex optimization are heavily used in a multitude of control system design and analysis tools. Recent surveys [39, 76] showed that such numerical tools had changed the control system design philosophy.

Nevertheless, linear and convex programming are limited in their ability to problems with specific structures. In several hybrid system design and verification problems, constraints are neither linear nor convex. This calls for efficient solvers that can reason about *general multivariate polynomial constraints*. In that regard, Cylindrical Algebraic Decomposition (CAD) has long been one of the most influential algorithms capable of solving general multivariate polynomial constraints. The first CAD algorithm was introduced by [21]. However, the use of CAD is often limited by the number of variables in the input polynomials, a reflection of its worst-case complexity that grows in a doubly exponential fashion in the number of variables [31].

To alleviate the CAD's doubly exponential issue, we introduce PolyAR, a highly parallelizable solver that uses convex programming and abstraction refinement to solve general multivariate polynomial inequality constraints. The main novel contributions of this chapter can be summarized as follows:

1. We propose PolyAR, a highly parallelizable solver that uses a combination of convex programming and abstraction refinement to solve multivariate polynomial inequality constraints.

2. PolyAR uses a novel convex abstraction refinement process where the original problem is iteratively relaxed into a series of convex programming problems with the aim to find

the solution and prune the search space. Second, it refines such abstraction where it becomes tighter with each iteration of the algorithm. Finally, it examines in parallel all the identified small volume regions left from the abstraction using off-the-shelf solvers (e.g., Z3 and Yices) to search for a solution in these regions.

3. We validate our approach by comparing the scalability of the proposed PolyAR solver with respect to the latest versions of state-of-art non-linear real arithmetic solvers, such as Z3 8.9 and Yices 2.6, on synthesizing stabilizing static output feedback controller (SOF) for linear time-invariant (LTI) continuous systems and designing non-parametric controller for the non-linear Duffing oscillator.

4. We demonstrate the performance of PolyAR on the problem of designing switching signals for continuous-time linear switching systems.

The remainder of the chapter is organized as follows: After the problem formulation in Section 1.2, we present the abstraction refinement of higher order polynomials using quadratic polynomials in Section 1.3. In Section 1.4, we introduce the the algorithm architecture of PolyAR. In Section 1.5, we provide the extension of multivariate polynomial constraints to a Satisfiability Modulo Theory (SMT) solving. Experimental results are given in Section 1.6.

Related work: The original CAD algorithm that was introduced by [21] was the first algorithm that solves general polynomial inequality constraints. However, due to Collins CAD's high time complexity, there have been improvements to this algorithm. [52] proposed an improvement of the projection operator in Collins CAD. However, the execution time of the modified version of the algorithm is still limited by the number of variables. [68] introduced a new projection operator which is a subset of Hong projection operator, removing redundant polynomials. However, McCallum proved that lifting over a sign-invariant CAD with this projection set is not sufficient to guarantee sign-invariance which makes the algorithm prone to error. The ABsolver tool proposed by [9] leverages a generic nonlinear optimization tool for solving non-linear constraints. However, generic optimization tool may produce incomplete results, and possibly incorrect, due to the local nature of the solver. [23] introduced Z3 which is another solver that implements an efficient nonlinear real arithmetic solver, that provide support for nonlinear polynomial arithmetic. However, it is still affected a lot by the increase in the number of variables in the polynomials. Because of the high complexity of existing approaches, we propose a highly parallelizable, efficient, and complete solver that uses the advantage and the simplicity of convex optimizations and abstraction refinement to solve higher order polynomial inequality constraints. To the best of our knowledge, this approach is new and has not been highlighted before.

1.2 Problem Formulation

1.2.1 Notation

We denote by $x = \big(x_1, x_2, \cdots, x_n\big) \in \mathbb{R}^n$ the set of real-valued variables, where $x_i \in \mathbb{R}$. We denote by $I_n = \big[\underline{d}_1, \overline{d}_1\big] \times \cdots \times \big[\underline{d}_n, \overline{d}_n\big] \subset \mathbb{R}^n$ the n-dimensional region. We denote the space of polynomials with n variables and coefficients in $\mathbb{R}$ by $\mathbb{R}[(x_1, x_2, \cdots, x_n)]$. We denote by $\wedge$ the Boolean conjunction. A set of the form $L_0^-(f) = \{\big(x_1, \cdots, x_n\big) \big| f\big(x_1, \cdots, x_n\big) \leq 0\}$ ($L_0^+(f) = \{\big(x_1, \cdots, x_n\big) \big| f\big(x_1, \cdots, x_n\big) \geq 0\}$) is called zero sublevel (superlevel) set of f, respectively.

1.2.2 Main Problem

In this chapter, we focus on *polynomial inequality constraints* with input ranges as closed boxes which are described in the following definition:

Definition 1.1. *A polynomial inequality constraint $F = I^n \ \wedge \ P_m$ consists of:*

- *a set of interval constraints:*

$$I^n = \bigwedge_{i=1}^{n} x_i \in [\underline{d}_i, \overline{d}_i], \tag{1.1}$$

- *a polynomial constraint:*

$$P_m = \bigwedge_{i=1}^{m} \; p_i\left(x_1, \cdots, x_n\right) \;\; \leq \;\; 0, \tag{1.2}$$

where $p_i\left(x\right) = p_i\left(x_1, \cdots, x_n\right) \in \mathbb{R}[(x_1, x_2, \cdots, x_n)]$ is a polynomial over variables $x_1, \cdots, x_n$. Without loss of generality, $\bigwedge_{i=1}^{m} \; p_i\big(x\big) \; \geq \; 0$ and $\bigwedge_{i=1}^{m} \; p_i\left(x\right) \; = \; 0$ can be encoded in constraint number (1.2).

We are now in a position to state the problem that we will consider in this chapter.

Problem 1.1. $\exists x = (x_1, \cdots, x_n)$ *subject to* $F = I^n \; \wedge \; P_m$.

1.3 Abstraction Refinement of Higher Order Polynomials Using Quadratic Polynomials

Traditional techniques for solving Problem 1.1 focus on finding all the n roots of the m polynomials and check all the regions between two successive roots to assign a positive/negative sign for each of these regions. Therefore, solving Problem 1.1 is known to be a doubly combinatorial problem in n with a total running time that is bounded by $\left(md\right)^{2^n}$ [31], where d is the maximum degree among polynomials in P_m.

In problems that are doubly exponential in the input space n, it is beneficial to isolate subsets of the search space in which the solution is guaranteed not to exist. Recall that Problem 1.1 asks for an x in $\mathbb{R}^n$ for which all the polynomials are negative. Therefore, a solution does not exist in subsets of $\mathbb{R}^n$ at which one of the polynomials is always positive. Similarly, isolating regions of the input space for which some of the polynomials are negative is also beneficial to finding the solution faster.

Our tool's main novelty is to use "convex abstractions" of the polynomials to find subsets of $L_0^+(p)$ and $L_0^-(p)$ efficiently. Indeed such "abstractions" may not be able to identify all regions for which the polynomial is positive or negative, which calls for an "abstraction refinement" process in which these "convex abstractions" become tighter with each iteration of the algorithm.

Figure 2.1(top) visualizes the proposed abstraction refinement process. Starting from a polynomial $p(x) \in \mathbb{R}[x]$ and an interval $I_n \subset \mathbb{R}^n$, we compute two quadratic polynomials:

$$O_1^p(x) \geq p(x), \quad U_1^p(x) \leq p(x), \qquad \forall x \in I_n,$$

where O and U stands for Over-approximate and Under-approximate quadratic polynomials, respectively, and the subscript in $O_1^p(x)$ and $U_1^p(x)$ encodes the iteration index of the abstraction refinement process. Computing such upper and lower abstractions can be carried out efficiently using Taylor approximation. Please refer to the example depicted in Figure 2.1 (top) for a visualization of $O_1^p(x)$ and $U_1^p(x)$ for one dimensional higher order polynomial (order ≥ 3) defined in the closed interval $[\underline{d},\ \overline{d}] \ \subset\ \mathbb{R}$.

It is clear from Figure 2.1(top) that the abstractions $O_1^p(x)$ and $U_1^p(x)$ fails to identify all subsets of $L_0^-(p)$ and $L_0^+(p)$. Therefore, the next step is to compute tighter over and under approximations of $p(x)$. Such a refinement process can be carried out by removing the zero superlevel and the zero sublevel sets, i.e., $L_0^+(U_1^p)$ and $L_0^-(O_1^p)$, identified using the previous abstraction and computing new over and lower approximation, as shown in Figure 2.1(bottom). The process of abstraction refinement can continue until the remaining subsets of the search space, in which case we call them ambiguous regions, and with some

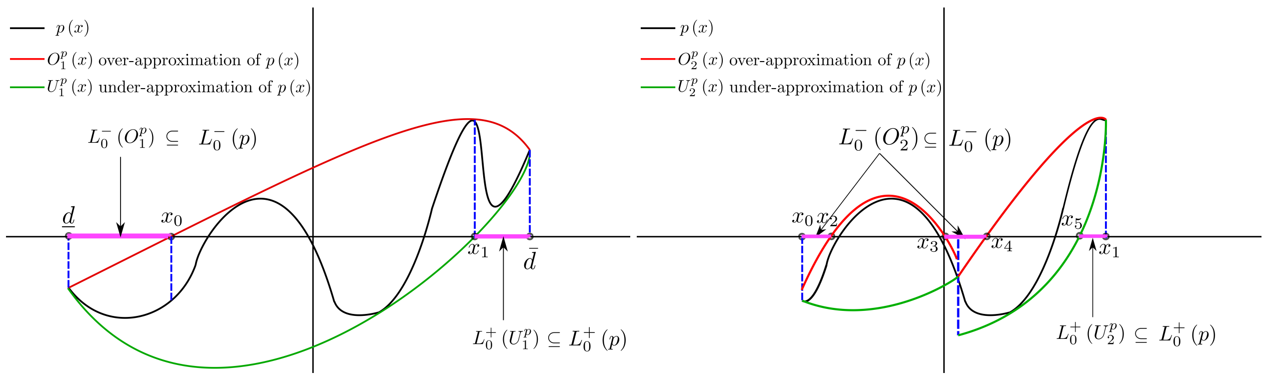


Figure 1.1: Abstraction Refinement of higher order polynomial using quadratic approximations: (top) first iteration and (bottom) second iteration.

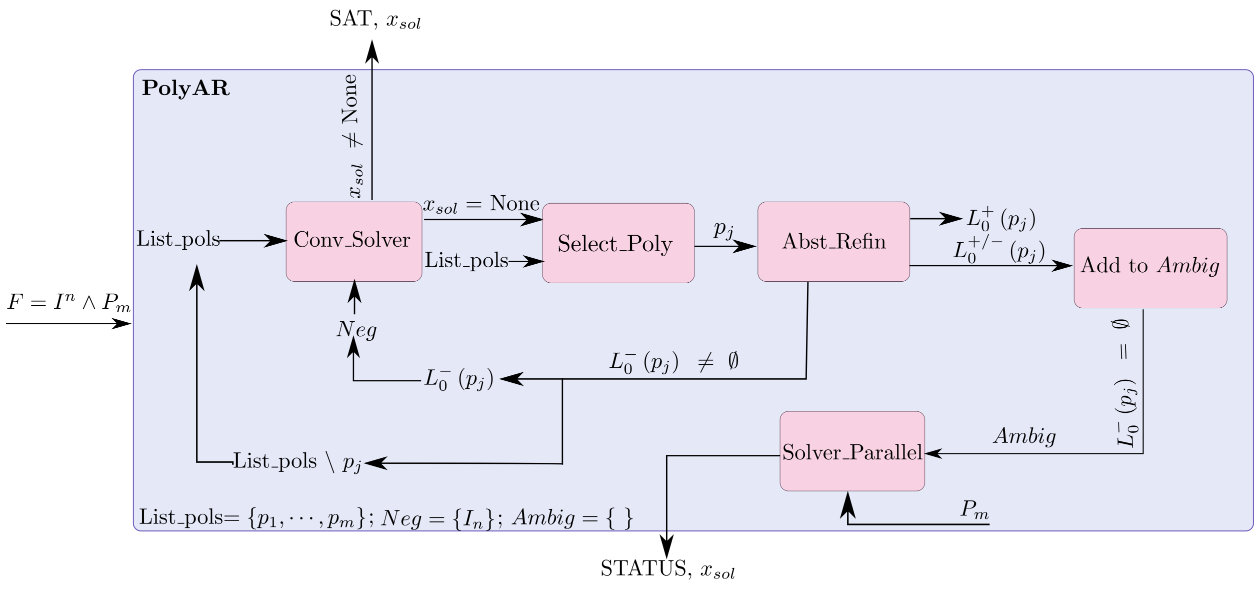


Figure 1.2: Framework of PolyAR.

abuse of notation, denoted them by $L_0^{+/-}(p)$, are *small* enough to be analyzed using off-the-shelf solvers. More details about the proposed abstraction refinement process are given in the next section.

1.4 Algorithm Architecture

In this section, we describe the different steps used by our solver PolyAR to solve Problem 1.1.

Our design methodology for the PolyAR tool aims to reduce the number of the required abstraction refinement and tries to find a solution early on in the process. To that end, the tool starts by computing a set of convex (quadratic or linear) polynomials $O_0^{p_i}(x), i = 1, \ldots, m$, that over approximate the original polynomials. The next step is to solve a convex feasibility problem aiming to find a solution that satisfy the constraints:

$$\exists x = (x_1, \ldots, x_n) \qquad \text{s.t.} \qquad O_0^{p_i}(x) \leq 0, \quad i = 1, \ldots, m.$$

Indeed, if such a convex problem is feasible, the tool terminates and returns the solution found by the convex feasibility problem above (**Conv_Solver**, Line 5 in Algorithm 1). If not, then the tool selects one polynomial p_j (**Select_Poly**, Line 10) to perform the abstraction refinement process. Indeed, several heuristics can be applied to select which polynomial will be selected. In the PolyAR tool, we opt-out to select the polynomial with the highest Lipschitz constant. Our intuition is that the higher the Lipschitz constant, the harder to obtain a tight over-approximation that can be used to find the solution.

Once a polynomial p_j is selected, the next step is to apply the abstraction refinement process on p_j (**Abst_Refin**, Line 11 in Algorithm 1). The objective of the abstraction refinement process is to identify subsets of the positive regions $L_0^+(p_j)$ and negative regions $L_0^-(p_j)$. Indeed, such abstraction refinement may not be able to identify all positive and negative regions, and hence a remaining portion of the search space may not be identified to belong to either $L_0^+(p_j)$ or $L_0^-(p_j)$ in which case it belongs to the ambiguous region $L_0^{+/-}(p_j)$. The abstraction refinement process of the polynomial p_j ensure that the volume of such ambiguous regions are below a certain user defined threshold.

The process of using the convex solver to find the solution and abstracting one polynomial continues. Since a solution of Problem 1.1 needs to lie in a negative region for all the polynomials, we confine the tool attention to the negative regions identified by the abstraction refinement in the previous iterations (Line 16 in Algorithm 1) to accelerate the process of searching for the solution.

While excluding the positive regions identified in previous iterations does not affect the tool (since a solution is guaranteed not to exist in such regions), excluding the ambiguous regions from the next iterations may affect the correctness of the tool. Therefore, the last step in the PolyAR tool is to examine all the identified ambiguous regions using off-the-shelf solvers (e.g., Z3 and Yices) to search for a solution in these regions (**Solver_Parallel**, Line 20 in Algorithm 1). Because the volume of these ambiguous regions is smaller than a user-defined threshold, the execution time of running off-the-shelf tools on such small volume regions is shorter than solving the original problem. This reflects that the number of roots for each polynomial is limited in small regions. Moreover, searching for a solution in these ambiguous regions can be highly parallelized, leading to an extra level of efficiency. This process is summarized in Algorithm 1 and Figure 1.2. We describe in detail each block algorithm that constitutes Algorithm 1 in the next subsections.

1.4.1 Early Termination Using Conv_Solver:

The objective of the **Conv_Solver** (Algorithm 2) is to search for a solution to Problem 1 using the information of (i) a set of closed convex regions Neg identified by the previous iterations of the abstraction refinement process and (ii) a list of polynomials (List_pols) that have not yet been processed by the abstraction refinement process.

Our approach is to compute a convex over-approximation of the polynomials in List_pols using Taylor approximation. To that end, we recall the definition of Taylor polynomials:

Definition 1.2. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be two times differentiable in open interval around a point $a \in \mathbb{R}^n$, then $f(x)$ can be written in terms of first and second order Taylor polynomials ,*

Algorithm 1 PolyAR(F)

Input: $F = I^n \wedge P_m$
Output: STATUS, x_{Sol}

```
1: $Neg = \{I_n\}$
2: $Ambig = \{\}$
3: List_pols $= \{p_1, \ldots, p_m\}$
4: while List_pols $\neq \emptyset$ do
5:     $x_{\text{Sol}} :=$ Conv_Solver $(Neg, \text{List\_pols})$
6:     if $x_{\text{Sol}} \neq$ None then
7:         STATUS=SAT
8:         return STATUS, $x_{\text{Sol}}$
9:     end if
10:    $p_j =$ Select_Poly (List_pols)
11:    $L_0^-(p_j), L_0^+(p_j), L_0^{+/-}(p_j)$
                                    $:=$ Abst_Refin $(Neg, p_j)$
12:    $Ambig.\text{add}\left(L_0^{+/-}(p_j)\right)$
13:    if $L_0^-(p_j) == \emptyset$ then
14:        break
15:    end if
16:    $Neg = L_0^-(p_j)$
17:    List_pols = List_pols $\setminus p_j$
18: end while
19: if List_pols $\neq \emptyset$ then
20:    STATUS, $x_{\text{Sol}} :=$ Solver_Parallel $(Ambig, P_m)$
21:    return STATUS, $x_{\text{Sol}}$
22: else
23:    STATUS=SAT
24:    $x_{\text{Sol}} = \text{center}(Neg)$
25:    return STATUS, $x_{\text{Sol}}$
26: end if
```

$T_1(x)$ and $T_2(x)$, around the neighborhood of a, as follows:

$$\begin{aligned} f(x) &= (x-a)^T D_f(a) + R_1(c_1) \\ &= T_1(x) + R_1(c_1), \end{aligned} \tag{1.3}$$

$$\begin{aligned} f(x) &= (x-a)^T D_f(a) \\ &+ \frac{1}{2}(x-a)^T H_f(a)(x-a) + R_2(c_2) \\ &= T_2(x) + R_2(c_2), \end{aligned} \tag{1.4}$$

where $T_1(x) = (x-a)^T D_f(a)$ and $T_2(x) = (x-a)^T D_f(a) + \frac{1}{2}(x-a)^T H_f(a)(x-a)$. $D_f(a)$ and $H_f(a)$ denote the Gradient vector and the Hessian matrix of f at the point a. $R_1(c_1)$ and $R_2(c_2)$ are reminders that depend on a and two points $c_1 \in \mathbb{R}^n$ and $c_2 \in \mathbb{R}^n$ that are located in the neighborhood of a. We upper-bound $R_1(c_1)$ and $R_2(c_2)$ by $M_1 \in \mathbb{R}_+$ and $M_2 \in \mathbb{R}_+$, i.e., $|R_1(c_1)| \leq M_1$ and $|R_2(c_2)| \leq M_2$. Hence:

$$T_1(x) - M_1 \leq f(x) \leq T_1(x) + M_1, \quad T_2(x) - M_2 \leq f(x) \leq T_2(x) + M_2.$$

Next, we check the convexity of the obtained second Taylor approximation and use it to compute the over-approximation function O^{p_i} whenever it is convex (Line 4 in Algorithm 2). Otherwise, we use the first Taylor approximation instead (Line 6 in Algorithm 2). Finally, for each $region$ in the set of negative regions (Neg), we solve the following convex feasibility problem:

$$x_{\text{Sol}} := \underset{x \in region}{\arg\min}\, 1 \quad \text{s.t.} \quad O^{p_i}(x) \leq \; 0, \quad i \in \text{List\_pols}. \tag{1.5}$$

Algorithm 2 Conv_Solver $(Neg, \text{List\_pols})$

```
Input: $Neg$, List_pols
Output: $x_{\text{Sol}}$

 1: for $region \in Neg$ do
 2:    for $i \in$ List_pols do
 3:       if $\texttt{Taylor}_{over}(p_i(x), 2)$ is convex then
 4:          $O^{p_i}(x) = \texttt{Taylor}_{over}(p_i(x), 2)$
 5:       else
 6:          $O^{p_i}(x) = \texttt{Taylor}_{over}(p_i(x), 1)$
 7:       end if
 8:    end for
 9:    $x_{\text{Sol}} := \arg\min_{x \in region} 1 \quad s.t. \quad O^{p_i}(x) \leq 0$ (see eq. (1.5)))
10:    return $x_{\text{Sol}}$
11: end for
```

1.4.2 Abstraction Refinement Using Abst_Refin:

Given the set of negative regions Neg identified by the previous abstraction refinement process along with the polynomial p_j selected by the **Select_Poly** algorithm, the objective of the **Abst_Refin** algorithm is to find subsets of the zero sublevel sets of p_j that lie inside Neg. The output of this algorithm are subsets of $L_0^-(p_j)$ and $L_0^+(p_j)$. The remainder of Neg is then considered to be part of the ambiguous regions $L_0^{+/-}(p_j)$. To do so, for every *region* in Neg, the tool initiates a list of ambiguous regions *List_Ambig_reg*, which will contain all the ambiguous regions from the abstraction refinement (Line 4 in Algorithm 3). Next, it selects one element from these ambiguous regions (Line 5 in Algorithm 3) and performs the abstraction refinement on this region iteratively until the volume of the remaining ambiguous region is smaller than a user-defined threshold (Line 6 in Algorithm 3). During the iterative abstraction refinement, all the identified zero sublevel and superlevel subsets are stored in the sets $L_0^-(p_j)$ and $L_0^+(p_j)$, respectively.

While the zero sublevel (superlevel) sets of the quadratic over-approximation (under-approxi mation) are ellipsoid or hyperboloid in general, we opt to represent all the subsets of $L_0^-(p_j)$

and $L_0^+(p_j)$ as $n-$dimensional hypercubes. This choice reflects the fact that off-the-shelf solvers (e.g., Z3 and Yices) can exploit the geometry of hypercubes to accelerate their computations. The process of finding these hypercubes can be summarized as follows:

1. **Step 1:** Compute the largest polytope inside the ellipsoid or hyperboloid representing the zero sublevel (superlevel) sets of the quadratic over-approximation (under-approximation) of p_j. To that end, we use a set of user-defined templates for the polytope.

2. **Step 2:** The previous step uses user-defined templates to find the polytope, such templates may fail and return an infeasible solution. In such scenarios, we split the ambiguous region into two (along the longest dimension) until a polytope is found.

3. **Step 3:** Finally, we under approximate the computed polytope with hypercubes.

This process is visualized in Figure 1.3. The details of each of these steps are given in the following subsections.

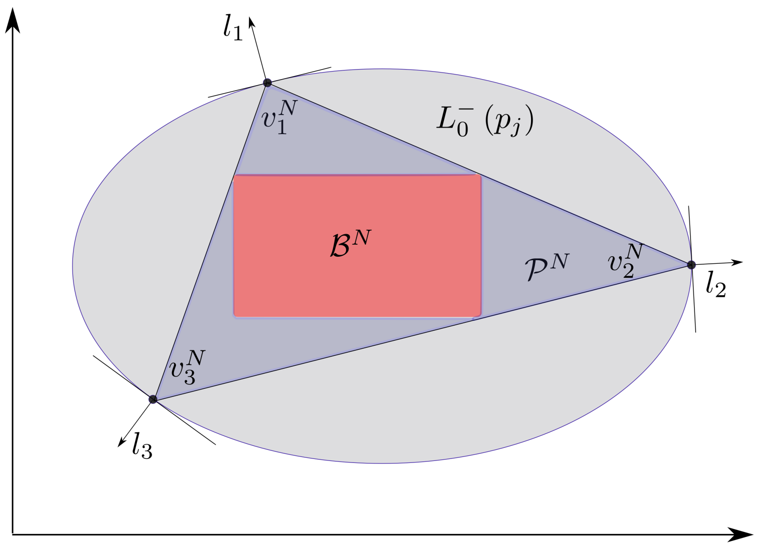


Figure 1.3: Polytopic under-approximation of a 2−dimensional ellipse sublevel set $L_0^-(p_j)$. $\mathcal{P}^N$ presents the under-approximate polytope inscribed in $L_0^-(p_j)$, and $\mathcal{B}^N$ represents the axis-aligned box of maximum volume inscribed in $\mathcal{P}^N$.

Step 1: Computing the largest polytope subset of $L_0^-(p_j)$ and $L_0^+(p_j)$

Given the over-approximation O^{p_j} computed using Taylor polynomials (detailed in Section 4.1) and a convex ambiguous region (Ambig_reg), we start by computing a set of $n+1$ vertices $v_1^N, \ldots, v_{n+1}^N$ that are inscribed in the ambiguous region *Ambig_reg*. Each vertex can be computed by solving the following convex optimization problem:

$$v_i^N = \operatorname*{arg\,min}_{v_i \in Ambig\_reg} \left(l_i^T v_i \right) \qquad \text{s.t.} \qquad O^{p_j}(v_i) \ \leq \ 0, \tag{1.6}$$

where l_i is a user defined normal vector (or template) (see Figure 1.3 for graphical representation of such normal vectors). Using these vertices, we can obtain the polytope $\mathcal{P}^N$ as:

$$\mathcal{P}^N = \textbf{Convex\_Hull} \left(v_1^N, \ldots, v_{n+1}^N \right).$$

Thanks to the constraints in the optimization problem (1.6) along with the convexity of $L_0^-(p_j)$, it is direct to conclude that the polytope $\mathcal{P}^N$ satisfy $\mathcal{P}^N \subset L_0^-(p_j)$. We compute the polytope $\mathcal{P}^P \subset L_0^+(p_j)$ in a similar fashion using the under-approximation U^{p_j} (Line 23 in Algorithm 3).

Step 3: Under approximate the polytopes with axis aligned boxes:

To compute the largest axis-aligned hypercube $\mathcal{B}^N$ inscribed inside the polytope $\mathcal{P}^N$, we solve the following convex optimization problem [10]:

$$\begin{aligned} &\operatorname*{arg\,max}_{(l_1^N, u_1^N, \ldots, l_n^N, u_n^N) \in \mathbb{R}^{2n}} \sum_{k=1}^{n} \log \left(u_k^N - l_k^N \right) \\ &\text{s.t.} \sum_{k=1}^{n} \left(p_{ik}^{N,+} u_k^N - p_{ik}^{N,-} l_k^N \right) \leq c_i^N, \ i = 1, \cdots, n_p, \end{aligned} \tag{1.7}$$

Algorithm 3 **Abst_Refin** (Neg, p_j)

Input: Neg, p_j
Output: $L_0^-(p_j)$, $L_0^+(p_j)$, $L_0^{+/-}(p_j)$

```
1: $L_0^-(p_j) = \{\ \}$, $L_0^+(p_j) = \{\ \}$, $L_0^{+/-}(p_j) = \{\ \}$
2: for $region \in Neg$ do
3:     $\text{vertices}^N = \{\ \}$, $\text{vertices}^P = \{\ \}$
4:     $List\_Ambig\_reg = \{region\}$
5:     $Ambig\_reg = \textbf{Select\_region}(List\_Ambig\_reg)$
6:     while $\text{Volume}(Ambig\_reg) > \text{Vol}_{\text{threshold}}$ do
7:         $List\_Ambig\_reg = List\_Ambig\_reg \setminus Ambig\_reg$
8:         for $i \in (1, \cdots, n+1)$ do
9:             $v_i^N = \underset{v_i \in Ambig\_reg}{\arg\min} \left(l_i^T v_i\right) \quad s.t. \quad O^{p_j}(v_i) \leq 0.$
10:            if $v_i^N \neq$ None then
11:                $\text{vertices}^N.\text{add}\left(v_i^N\right)$
12:            end if
13:            $v_i^P = \underset{v_i \in Ambig\_reg}{\arg\min} \left(l_i^T v_i\right) \quad s.t. \quad U^{p_j}(v_i) \leq 0.$
14:            if $v_i^P \neq$ None then
15:                $\text{vertices}^P.\text{add}\left(v_i^P\right)$
16:            end if
17:        end for
18:        if $\left(\text{vertices}^N == \emptyset \text{ and } \text{vertices}^P == \emptyset\right)$ then
19:            $Ambig\_reg_1$, $Ambig\_reg_2$
                                   $:= \textbf{Half\_Div}(Ambig\_reg)$
20:            $List\_Ambig\_reg.add(Ambig\_reg_1, Ambig\_reg_2)$
21:        else if $\left(\text{vertices}^N \neq \emptyset \text{ and } \text{vertices}^P \neq \emptyset\right)$ then
22:            $\mathcal{P}^N = \textbf{Convex\_Hull}\left(\text{vertices}^N\right)$
23:            $\mathcal{P}^P = \textbf{Convex\_Hull}\left(\text{vertices}^P\right)$
24:            $\mathcal{B}^N = \textbf{Box}\left(\mathcal{P}^N\right)$; $\mathcal{B}^P = \textbf{Box}\left(\mathcal{P}^P\right)$
25:            $L_0^-(p_j).\text{add}\left(\mathcal{B}^N\right)$; $L_0^+(p_j).\text{add}\left(\mathcal{B}^P\right)$
26:            $Ambig\_reg = Ambig\_reg \setminus \left(\mathcal{B}^N \cup \mathcal{B}^N\right)$
27:            $List\_Ambig\_reg.\text{add}\left(Ambig\_reg\right)$
28:        end if
29:        $Ambig\_reg = \textbf{Select\_region}(List\_Ambig\_reg)$
30:    end while
31:    $L_0^{+/-}(p_j).add(List\_Ambig\_reg)$
32: end for
33: return $L_0^-(p_j)$, $L_0^+(p_j)$, $L_0^{+/-}(p_j)$
```

where $(l_1^N, u_1^N, \ldots, l_n^N, u_n^N) \in \mathbb{R}^{2n}$ is the representation of the box $\mathcal{B}^N$ with (l_k, u_k) is the lower/upper limit of the box in the kth dimension, $p_{ik}^{N,+} = \max\{p_{ik}^N, 0\}$, $p_{ik}^{N,-} = \max\{-p_{ik}^N, 0\}$,

Algorithm 4 Solver_Parallel $(Ambig, P_m)$

```
Input: $Ambig$, $P_m$
Output: STATUS, $x_{\text{Sol}}$
1: The tool runs off-the-shelf solvers such as Z3 or Yices on small-volume ambiguous regions
   in $Ambig$ in parallel:
2: STATUS, $x_{\text{Sol}}$ := Z3/Yices_Parall $(Ambig, P_m)$
3: return STATUS, $x_{\text{Sol}}$
```

and p_{ik}^N, c_i^N are the rows of the half-space matrix/vector representation of the polytope $\mathcal{P}^N$.

1.4.3 Highly Parallelizable Analysis of Ambiguous Regions using Solver_Parallel

Once all the ambiguous regions are identified, the next step is to analyze all of them using off-the-shelf solvers. In particular, PolyAR supports the use of the latest versions Z3 8.9 and Yices 2.6 solvers. Thanks to the fact that all the ambiguous regions are hypercubes, both these solvers can exploit the geometry of the region to accelerate their computations. Also, thanks to the fact that the volume of all ambiguous regions is lower than a user-defined threshold, the CAD algorithm can run efficiently. To that end, PolyAR tool runs multiple instances of Z3 or Yices to analyze all these ambiguous regions in parallel as summarized in Algorithm 4.

1.5 Extension to SMT solving

We extend the PolyAR solver described in the previous sections to account for combinations of Boolean and Polynomial inequality constraints of the form:

$$\begin{aligned}
&\exists (b_1, \ldots, b_o, x_1, \ldots, x_n) \in \mathbb{B}^o \times \mathbb{R}^n, && \\
&\quad \text{subject to:} && \\
&p_i(x_1, \ldots, x_n) \leq 0, && i = 1, \ldots, m \qquad (1.8)\\
&x_k \in [\underline{d}_k, \overline{d}_k], && k = 1, \ldots, n \qquad (1.9)\\
&\varphi_j(b_1, \ldots, b_o) \longleftrightarrow \texttt{TRUE}, && j = 1, \ldots, r \qquad (1.10)\\
&b_l \longleftrightarrow \big(p_{l+m}(x_1, \ldots, x_n) \ \leq \ 0\big), && l = 1, \ldots, h \qquad (1.11)
\end{aligned}$$

where $\varphi_j(b_1, \ldots, b_o)$ is any combinations of Boolean and pseudo-Boolean predicates.

We can create a Satisfiability Modulo Theory (SMT) solver by combining a SAT solver for Boolean and pseudo-Boolean constraints and a theory solver (PolyAR) for interval and polynomial constraints on real numbers by following the lazy SMT paradigm [7]. The SAT solver solves the combination of Boolean and pseudo-Boolean constraints using the David-Putnam-Logemann-Loveland (DPLL) algorithm and suggests satisfying assignments for the Boolean variables b and thus suggesting which polynomial constraints should jointly satisfied (or unsatisfied). The theory solver (PolyAR) checks the validity of the given assignments and provides an explanation of the conflict, i.e., an *UNSAT certificate*, whenever a conflict is found. Each certificate is a new Boolean constraint that will be used by the SAT solver to prune the search space.

While in the lazy SMT paradigm, the PolyAR solver needs to be executed multiple times with a different set of polynomial constraints, we modify the PolyAR solver to perform all the abstraction refinement for all the polynomials as a pre-processing step. This eliminates the need to re-compute the same abstraction refinement every time the PolyAR solver is executed.

Whenever the SAT solver assigns one of the Boolean variables b_l in (1.11) to zero, then the PolyAR solver needs to guarantee that the corresponding polynomial p_{l+m} satisfy $p_{l+m}(x) > 0$ or equivalently $-p_{l+m}(x) \leq 0$. To eliminate the need to apply the convex abstraction refinement process for both $p_{l+m}(x)$ and $-p_{l+m}(x)$, the PolyAR solver computes the negative and positive boxes ($\mathcal{B}^N$ and $\mathcal{B}^P$) only for $p_{l+m}(x)$ and flips their usage for $-p_{l+m}(x)$.

1.6 NUMERICAL RESULTS

In this section, we compare the performance of PolyAR to the state-of-the-art solvers Z3 8.9 and Yices 2.6. The objective of this comparison is to study the performance on:

- Problems that appear naturally in parametric controller synthesis. In particular, we focus on the problem of designing stabilizing SOF controllers for LTI systems [4].
- Problems that appear in non-parametric controller synthesis for non-linear systems. In particular, we focus on the problem of designing a controller for the nonlinear Duffing oscillator [42].
- Additionally, we demonstrate the performance of PolyAR on designing a hybrid switching system; a problem which state-of-the-art tools are incapable of handling.

All the experiments were executed on an Intel Core i7 2.6-GHz processor with 16 GB of memory.

1.6.1 Static Output Feedback Controller Synthesis for Linear Time Invariant Systems

In this subsection, we assess the scalability of the PolyAR solver compared to state-of-the-art solvers on control synthesis problems. In particular, we consider the problem of synthesizing a parametric controller for the following continuous LTI system:

$$\dot{x} = Ax + Bu, \qquad y = Cx,$$

where $x \in \mathbb{R}^{n_A}$ is the system state, $u \in \mathbb{R}^{n_B}$ is the system control input, $y \in \mathbb{R}^{n_C}$ is the system output, and the matrices $A \in \mathbb{R}^{n_A \times n_A}$, $B \in \mathbb{R}^{n_A \times n_B}$ and $C \in \mathbb{R}^{n_C \times n_A}$ are the system matrices. We are interested in designing a static output feedback controller of the form:

$$u = Ky,$$

such that the resulting closed loop system:

$$\dot{x} = (A + BKC)x,$$

is stable, i.e., the matrix $A + BKC$ is Hurwitz.

We follow the steps detailed in [4] to pose the problem of designing the static output feedback controller as a set of polynomial constraints using the Routh-Hurwitz stability criteria. The Routh-Hurwitz stability criteria result in a set of n_A polynomials in the elements of the controller matrix K. We consider five instances of the controller synthesis problem with the following parameters:

- **Example 1**: $n_A = 3, n_B = 4, n_C = 4$ which results in 3 polynomial constraints with 16 variables and max polynomial order of 4. We restrict the elements of the controller matrix to be inside $[-4, 7]$.

- **Example 2**: $n_A = 3, n_B = 5, n_C = 5$ which results in 3 polynomial constraints with 25 variables and max polynomial order of 3. We restrict the elements of the controller matrix to be inside $[-0.5, 1]$.

- **Example 3**: $n_A = 2, n_B = 6, n_C = 6$ which results in 2 polynomial constraints with 36 variables and max polynomial order of 3. We restrict the elements of the controller matrix to be inside $[0, 5]$.

- **Example 4**: $n_A = 2, n_B = 7, n_C = 7$ which results in 2 polynomial constraints with 49 variables and max polynomial order of 2. We restrict the elements of the controller matrix to be inside $[-10, 0]$.

- **Example 5**: $n_A = 5, n_B = 4, n_C = 4$ which results in 5 polynomial constraints with 16 variables and max polynomial order of 4. We restrict the elements of the controller matrix to be inside $[-4, 7]$. In addition, we want to enforce the following controller structure:

  $$k_{21} \times k_{22} \times k_{23} < \ 0, \quad k_{21} + k_{22} + k_{23} < -1$$

  , which can be encoded using the additional SMT constraints:

  $$\begin{aligned} & b_1 \wedge b_2 \longleftrightarrow \text{True}, \\ & b_1 \rightarrow k_{21} \times k_{22} \times k_{23} < 0, \\ & b_2 \rightarrow k_{21} + k_{22} + k_{23} < -1, \end{aligned}$$

  where k_{ij} are the elements of the controller matrix K.

For each of these examples, we generate random system matrices from a zero-mean normal distribution and feed them to four versions of our solver PolyAR:

Table 1: Experiment results for SOF design. The timeout is set by 3600 s.

Example	Times (seconds)					
	Z3 8.9	Yices 2.6	PolyAR+Z3 (1 thread)	PolyAR+Z3 (max threads)	PolyAR+Yices (1 thread)	PolyAR+Yices (max threads)
1	*timeout*	*timeout*	*timeout*	7.552	**2.405**	2.442
2	*timeout*	*timeout*	83.776	114.453	*timeout*	**3.766**
3	*timeout*	*timeout*	23.551	23.970	*timeout*	**8.725**
4	*timeout*	*timeout*	0.718	0.729	**0.416**	0.432
5	*timeout*	*timeout*	3.636	3.768	0.621	**0.498**
# Problems Solved	0	0	4	5	3	**5**
Total Time (seconds)	*timeout*	*timeout*	111.681	150.472	3.442	**15.863**

- PolyAR + Z3 (1 thread): This version uses one instance of Z3 to analyze all the ambiguous regions.

- PolyAR + Z3 (max threads): This version uses a separate instance of Z3 to analyze each of the ambiguous regions. All Z3 instances are running in parallel.

- PolyAR + Yices (1 thread): This version uses one instance of Yices to analyze all the ambiguous regions.

- PolyAR + Yices (max thread): This version uses a separate instance of Yices to analyze each of the ambiguous regions. All Yices instances are running in parallel.

We compare the execution times of these four solvers with Z3 8.9 and Yices 2.6. Table 1 shows the execution time for all the solvers. As evident by the results in Table 1, off-the-shelf solvers are incapable of solving all the five examples and they time out after one hour. On the other hand, and thanks to the abstraction refinement process, the PolyAR solver is able to solve all the instances in a few seconds, leading to $240X$ speed up in the total execution time in the PolyAR+Yices (max threads) case, evidence of the scalability of the proposed approach.

In the following, we give the stabilizing controller matrices K_1, K_2, K_3, K_4, K_5, and the two

Boolean variables b_1 and b_2 that PolyAR (Yices) returned for Examples 1, 2, 3, 4, and 5:

$$K_1 = \begin{bmatrix} -4 & -2 & -2 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 4 & 1 & 1 \end{bmatrix}, K_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$K_3 = \begin{bmatrix} 1 & 1 & 3 & 3 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 & 2 \\ 2 & 0 & 0 & 2 & 2 & 5 \\ 0 & 0 & 2 & 2 & 0 & 2 \\ 2 & 2 & 0 & 0 & 2 & 5 \\ 5 & 2 & 2 & 2 & 2 & 2 \end{bmatrix}, K_4 = \begin{cases} -8, & i = 1, j = 1, \\ -5, & 2 \leq i, j \leq 7, \end{cases}$$

$$K_5 = \begin{bmatrix} -2 & 4 & -2 & 7 \\ 1 & -3.5 & 1 & 1 \\ 5 & 6 & 1 & 1 \\ 1 & 6.99 & 1 & 1 \end{bmatrix}, \ b_1 = \text{True}, \ b_2 = \text{True}.$$

It is easily to note that the solutions given by PolyAR (Yices) satisfies the two Boolean constraints, i.e., $k_{21} \times k_{22} \times k_{23} = -3.5 < \ 0$ and $k_{21} + k_{22} + k_{23} = -1.5 < -1$.

In conclusion, PolyAR+Yices (max thread) solver outperforms all the other solvers due to the effectiveness of Yices in reasoning about problems with small volumes.

1.6.2 Non-Linear Controller Design for a Duffing Oscillator

In this subsection, we assess the scalability of PolyAR solver compared to state-of-the-art solvers on synthesizing a non-parametric controller for a Duffing oscillator reported by [42].

The dynamics of the oscillator is given by the higher-order differential equation:

$$y^{(n)}(t)+\cdots+y^{(2)}(t)+2\zeta y^{(1)}(t)+y(t)+y(t)^{3}=u\left(t\right), \tag{1.12}$$

where $y \in \mathbb{R}$ is the continuous state variable and $u \in \mathbb{R}$ is the control input. The parameter ζ is the damping coefficient. The objective of the control is to regulate the state to the origin. To derive the discrete-time model, forward difference approximation is used (with sampling period of $h = 0.05$ time units). The resulting state space model with discrete state vector $x = [x_1, x_2, \cdots, x_n]^T = [y, y^{(1)}, \cdots, y^{(n-1)}]^T \in \mathbb{R}^{n-1}$ and input $u \in \mathbb{R}$ is:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}^{+} = \begin{bmatrix} 1 & h & 0 & \cdots & \cdots & 0 \\ 0 & 1 & h & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ -h & -2\zeta h & -h & \ldots & -h & 1-h \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ h \end{bmatrix} u + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ -hx_1^3 \end{bmatrix}. \tag{1.13}$$

The previous equation is written in the form of $x\left(k+1\right) = Ax\left(k\right) + Bu\left(k\right) + E\left(x\right)$, which includes a nonlinear term $E\left(x\right) = \begin{bmatrix} 0, \cdots, -hx_1^3\left(k\right) \end{bmatrix}^T$. Our objective is to design a non-parametric controller. To that end, we encode the controller as the solution of a feasibility problem of several constraints that capture the system dynamics, state/input constraints, and stability constraints as discussed below.

First, to enforce the stability of the resulting non-parametric controller, we consider the candidate quadratic Lyapunov function $V\left(x\right) = x^T Px$ with the symmetric positive definite matrix P is a solution of the discrete-time Lyapunov equation $APA^T + P + Q = 0$ and is a positive definite matrix. Thanks to the fact that $E(x)$ satisfies $\lim_{\|x\|\to 0} \frac{\|E(x)\|}{\|x\|} = 0$ along with the Lyapunov's indirect method in [58], one can directly conclude that $V(x)$ is indeed a Lyapunov function. For simplicity, we pick $Q = I_n$, where I_n is the identity matrix of size n.

Moreover, to ensure the smoothness of the resulting controller signals, we add additional filters in the form of high order polynomial $L(x, u) \leq 0$. In addition, we consider the state-constraints of the form $\|x(k)\|_\infty \leq 0.6$.

The final non-parametric controller is then encoded as the solution of the following feasibility problem:

$$\begin{aligned}
&\exists x_1(k), \ldots x_n(k), x_1(k+1), \ldots x_n(k+1), u(k) \\
&\text{subject to :} \\
&\qquad x(k+1) = Ax(k) + Bu(k) + E(x), \\
&\qquad V(x(k+1)) - V(x(k)) \leq -\epsilon, \\
&\qquad L(x(k), u(k)) \leq 0, \\
&\qquad \|x(k)\|_\infty \leq 0.6.
\end{aligned} \tag{1.14}$$

Since the PolyAR solver only handles polynomial inequalities, hence, we transform the equality constraint $x(k+1) = Ax(k) + Bu(k) + E(x)$ above into two inequalities $x(k+1) - Ax(k) + Bu(k) + E(x) \leq \epsilon \ \wedge \ x(k+1) - Ax(k) + Bu(k) + E(x) \geq -\epsilon$, where $\epsilon \in \mathbb{R}$ is a small value.

We consider three instances of the controller synthesis problem for the Duffing oscillator with the following parameters:

- $n = 2$, $\zeta = 0.3$, $x(0) = [0.4, 0.1]^T$, $L(x(k), u(k)) = x_1^{11}(k) + x_2^{11}(k) - u^{10}(k)$, which results in 6 polynomial constraints with 3 variables and max polynomial order of 11.
- $n = 3$, $\zeta = 1.0$, $x(0) = [0.1, 0.1, 0.1]^T$, $L(x(k), u(k)) = x_1^5(k) + x_2^5(k) + x_3^5(k) + u^5(k)$, which results in 8 polynomial constraints with 4 variables and max polynomial order of 5.

- $n = 4$, $\zeta = 1.75$, $x(0) = [0.1, 0.1, 0.01, 0.1]^T$, $L(x(k), u(k)) = x_1^4(k) + x_2^4(k) + x_3^4(k) + x_4^4(k) - u^4(k)$, which results in 10 polynomial constraints with 5 variables and max polynomial order of 4.

We feed the resultant polynomial inequality constraint to PolyAR+Yices, PolyAR+Z3, Yices, and Z3. We solve the feasibility problem for $n = 2$, $n = 3$, and $n = 4$. We set the timeout to be $1s$. Figure 2.8 (left) shows the state-space evolution of the controlled Duffing oscillator for different solvers for number of variables n of $2, 3$, and 4. Figure 2.8 (right) shows the evolution of the execution time of the solvers during the 20 seconds. As it can be seen from Fig. 4, our solver PolyAR+ Yices succeeded to find a control input u that regulates the state to the origin for all n. However, off-the-shelf solvers are incapable of solving all the three instances and they early time out after one second out of the simulated 20 *seconds*.

1.6.3 Designing Switching Signals for Continuous-Time Linear Switching Systems

In this subsection, we show how to use the PolyAR solver to successfully design a controller for a continuous-time linear switching system. In particular, we consider the following switching dynamics:

$$\dot{x} = A_{\sigma(t)}x, \qquad \sigma(t) = \{1, 2, 3\},$$

with $x(t) \in \mathcal{X} \subset \mathbb{R}^2$ is the system state at time t and the matrices A_1, A_2, and $A_3 \in \mathbb{R}^{2\times 2}$ represents three modes for the switching system. Consider the state space in Figure 1.5. The objective is to design a switching signal $\sigma(t)$ that can steer the state of the system to the goal set $\mathit{Goal} \subset \mathcal{X}$ while avoiding entering the obstacle set $\mathit{Obstacle} \subset \mathcal{X}$. For simplicity, we confine our attention to step-wise switching signals $\sigma(t)$. That is, we assume the switching signal $\sigma(t)$ will be constant for some amount of time $t_1, t_2, \ldots, t_L$. Our objective is then to

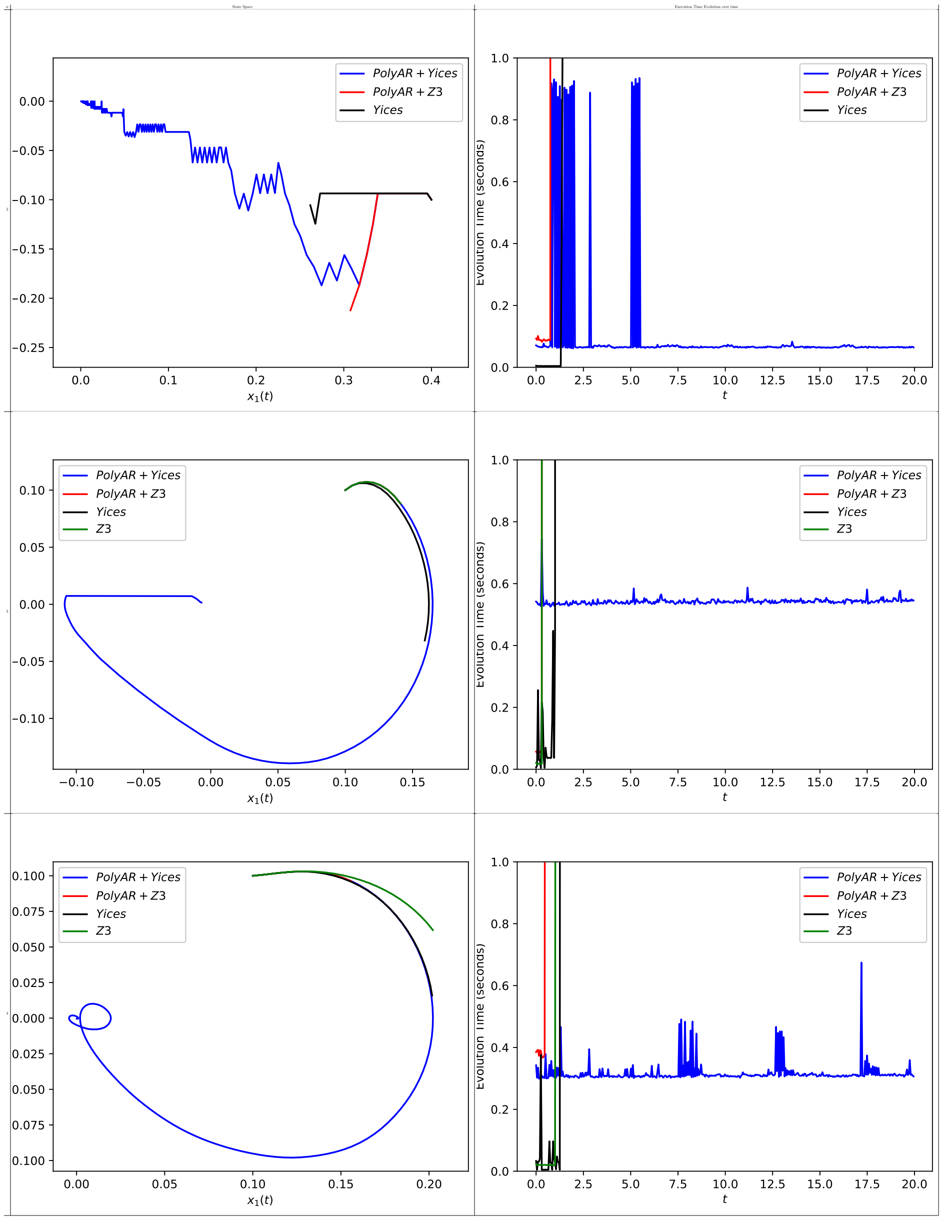


Figure 1.4: Results of controlling the Duffing oscillator with different n (left) evolution of the states $x_1(k)$ and $x_2(k)$ for the solvers in the state-space, (right) evolution of the execution time of solvers during the 20 seconds. The timeout is equal to $1s$. Trajectories are truncated once the solver exceeds the timeout limit.

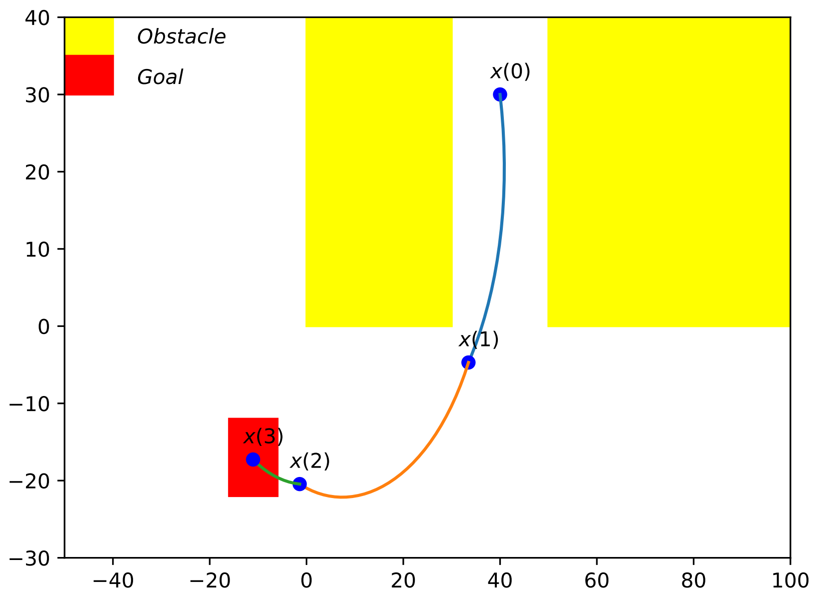


Figure 1.5: The trajectory that starts from an initial state $x(0) = [40, 30]^T$ and reaching a final state $x(3) \in \mathit{Goal}$ while avoiding the obstacles. The goal and the obstacles are represented with a red and yellow rectangle, respectively.

design the switching times and the associated system mode that leads to the satisfaction of the reach-avoid specifications. To that end, we define a set of Boolean variables b_{ij} such that b_{ij} is equal to 1 whenever the jth mode is active during t_i. Given the initial condition of the system $x(0)$, we can use these Boolean variables to encode the problem of designing the switching signal as the following SMT constraints:

$$
\begin{aligned}
&\exists b_{11}, \ldots, b_{13}, \ldots, b_{L1}, \ldots, b_{L3}, x(1), \ldots, x(L), t_1, \ldots, t_L \\
&\text{subject to:} \\
&\quad b_{11} \rightarrow x\left(1\right) = \exp\left(A_1 t_1\right)\, x\left(0\right), \\
&\quad b_{12} \rightarrow x\left(1\right) = \exp\left(A_2 t_1\right)\, x\left(0\right), \\
&\quad b_{13} \rightarrow x\left(1\right) = \exp\left(A_3 t_1\right)\, x\left(0\right), \\
&\quad b_{11} + b_{12} + b_{13} = 1, \\
&\qquad\qquad \vdots \\
&\quad b_{L1} \rightarrow x\left(L\right) = \exp\left(A_1 t_L\right)\, x\left(L-1\right), \\
&\quad b_{L2} \rightarrow x\left(L\right) = \exp\left(A_2 t_L\right)\, x\left(L-1\right), \\
&\quad b_{L3} \rightarrow x\left(L\right) = \exp\left(A_3 t_L\right)\, x\left(L-1\right), \\
&\quad b_{L1} + b_{L2} + b_{L3} = 1, \\
&\quad x\left(1\right), \ldots, x\left(L-1\right) \notin \mathit{Obstacle}, \\
&\quad x\left(L\right) \in \mathit{Goal},
\end{aligned}
\tag{1.15}
$$

where the constraint $b_{i1} + b_{i2} + b_{i3} = 1$ is a pseudo-Boolean constraint that enforces the consistency between the Boolean variables such that only one of the three modes A_1, A_2 and A_3 can be selected during the period $t_{j-1} < t \leq t_j$. Since PolyAR solver only handles polynomial inequalities, we approximate the exponential matrix $\exp\left(A_i t_j\right) \approx I_2 + t_j A_i + \frac{t_j^2 A_i^2}{2} + \frac{t_j^3 A_i^3}{6}$, $i, j = 1, \cdots, L$, where I_2 is the identity matrix of size 2 and $A_i^n = A_i \times \cdots \times A_i$. Furthermore, we transform the equality $x\left(i\right) = \exp\left(A_i t_j\right)\, x\left(i-1\right)$, $i, j = 1, \cdots, L$, into two inequalities $x\left(i\right) - \exp\left(A_i t_j\right)\, x\left(i-1\right) \leq \epsilon \wedge x\left(i\right) - \exp\left(A_i t_j\right)\, x\left(i-1\right) \geq -\epsilon$, where $\epsilon \in \mathbb{R}$ is a small value.

In our experiments, we pick the horizon $L = 3$ and the modes $A_1 = \begin{bmatrix} -1, 2 \\ -2, -2 \end{bmatrix}$, $A_2 =$

$\begin{bmatrix} -1,3 \\ -3,-1 \end{bmatrix}$, and $A_3 = \begin{bmatrix} 0,2 \\ -2,0 \end{bmatrix}$, and we start with an initial state $x(0) = [40,30]^T$. We restrict the states to be inside $\mathcal{X} \in [-100,100]$. We feed the resultant polynomial inequality constraint to PolyAR+Yices, and as it can be seen from Fig.1.5, the solver succeeded to find the right modes ($b_{11} = b_{22} = b_{33} = 1$) and the necessary times $t_1 = 0.391s$, $t_2 = 0.5s$, and $t_3 = 0.25s$ that ensures that $x(3)$ reaches a *Goal* while the intermediate states $x(2), x(1)$ avoid *Obstacles*. In addition, we remark that the trajectory between $x(0)$ and $x(2)$ is making its way to the equilibrium point $[0,0]^T$. This is explained by the fact that the matrices A_1 and A_2 are stables. Our solver computes the necessary time t_3 that ensures that the final state $x(3) \in$ *Goal* and does not converge to the equilibrium point.

Chapter 2

PolyARBerNN: A Neural Network Guided Solver and Optimizer for Bounded Polynomial Inequalities

In this chapter, we introduce a solver named PolyARBerNN, an enhancement to PolyAR introduced in chapter 1, which uses convex polynomials as abstractions for highly nonlinear polynomials. Such abstractions were previously shown to be powerful to prune the search space and restrict the usage of sound and complete solvers to small search spaces. Compared with the previous efforts on using convex abstractions, PolyARBerNN provides three main contributions namely (i) a neural network guided abstraction refinement procedure that helps selecting the right abstraction out of a set of pre-defined abstractions, (ii) a Bernstein polynomial-based search space pruning mechanism that can be used to compute tight estimates of the polynomial maximum and minimum values which can be used as an additional abstraction of the polynomials, and (iii) an optimizer that transforms polynomial objective functions into polynomial constraints (on the gradient of the objective function) whose solutions are guaranteed to be close to the global optima. These enhancements together allowed the PolyARBerNN solver to solve complex instances and scales more favorably compared to the state-of-art nonlinear real arithmetic solvers while maintaining the soundness and completeness of the resulting solver. In this chapter, we show in test benches that PolyARBerNN achieved 100X speedup compared with Z3 8.9, Yices 2.6, and PVS (a solver that uses Bernstein expansion to solve multivariate polynomial constraints) on a variety of standard test benches. In this chapter, we implement an optimizer called PolyAROpt that uses PolyARBerNN to solve constrained polynomial optimization problems. Numerical results show that PolyAROpt is able to solve high-dimensional and high order polynomial optimization problems with higher speed compared to the built-in optimizer in the Z3 8.9 solver.

2.1 Introduction

Constraint solvers and optimizers have been used heavily in the design, synthesis, and verification of cyber-physical systems [95, 104, 2, 13, 6, 105, 28, 67, 60, 80, 75]. Examples includes verification of neural network controlled autonomous systems [88], formal verification of human-robot interaction in healthcare scenarios [62], automated synthesis for distributed cyber-physical systems [79], design for cyber-physical systems under sensor attacks [84], air traffic management of unmanned aircraft systems [71], software verification for the next generation space-shuttle [73], and conflict detection for aircraft [72].

In this chapter, we will focus on the class of general multivariate polynomial constraints (also known as nonlinear real arithmetic). Multivariate polynomial constraints appear naturally in the design, synthesis, and verification of these systems. It is not then surprising that the amount of attention given to this problem in the last decade, as evidenced by the amount of off-the-self solvers that are designed to solve feasibility and optimization problems over general multivariate polynomial constraints, including Z3 [23], Coq [65], Yices [26], PVS [70], Cplex, [66], CVXOPT [94], and Quadprog [50]. Regardless of their prevalence in several synthesis and verification problems, well-known algorithms—that are capable of solving a set of polynomial constraints—are shown to be doubly exponential [31], placing a significant challenge to design efficient solvers for such problems.

Recently, neural networks (NNs) have shown impressive empirical results in approximating unknown functions. This observation motivated several researchers to ask how to use NNs to tame the complexity of NP-hard problems. Examples are the use of NNs to design scalable solvers for program synthesis [24], traveling salesman problem [11], first-order theorem proving [54], higher-order theorem proving [56], and Boolean satisfiability (SAT) problems [82]. While several of these solvers sacrifice either soundness or correctness guarantees, we are interested in this chapter on using such empirically powerful NNs to design a sound and complete solver for nonlinear real arithmetic.

In addition to NNs, polynomials constitute a rich class of functions for which several approximators have been studied. Two of the most famous approximators for polynomials are Taylor approximation and Bernstein polynomials. These two approximators have been successfully used in solvers like Coq and PVS [15, 70]. This opens the question of how to combine all those approximation techniques, i.e., NNs, Taylor, and Bernstein approximations, to come up with a scalable solver that can reason about general multivariate polynomial constraints.

We introduce PolyARBerNN, a novel sound and complete solver for polynomial constraints that combines these three function approximators (NNs, Taylor, and Bernstein) to prune the search space and produce small enough instances in which existing sound and complete solvers (based on the well-known Cylindrical Algebraic Decomposition algorithm) can easily reason about. In general, in this chapter, we provide the following contributions:

1. We introduce a novel NN-guided abstraction refinement process in which a NN is used to guide the use of Taylor approximations to find a solution or prune the search

space. We analyzed the theoretical characteristics of such a NN and provided empirical evidence on the generalizability of the trained NN in terms of its ability to guide the abstraction refinement process for unseen polynomials with various numbers of variables and orders.

2. We complement the NN-guided abstraction refinement with a state-space pruning phase using Bernstein approximations that accelerates the process of removing portions of the state space in which the sign of the polynomial does not change.

3. We validate our approach by first comparing the scalability of the proposed PolyARBerNN solver with respect to PVS, a library that uses Bernstein expansion to solve polynomial constraints. Second, we compared the execution times of the proposed tool with the latest versions of the state-of-the-art nonlinear arithmetic solvers, such as Z3 8.9, Yices 2.6 by varying the order, the number of variables, and the number of the polynomial constraints for instances when a solution exists and when a solution does not exist. We also compared the scalability of the solver against Z3 8.9 and Yices 2.9 on the problem of synthesizing a controller for a cyber-physical system.

4. We propose PolyAROpt, an optimizer that uses PolyARBerNN to solve constrained multivariate polynomial optimization problems. The theoretical analysis shows that PolyAROpt is capable of providing solutions that are ϵ close to the global optima (for any $\epsilon > 0$ chosen by the user). Numerical results show that PolyAROpt solves high-dimensional and high-order optimization problems with high speed compared to the built-in optimizer in Z3 8.9 solver. We also validate the effectiveness of PolyAROpt on the problem of computing the reachable sets of polynomial dynamical systems.

Related work: Cylindrical algebraic decomposition (CAD) was introduced by Collins [21] in 1975 and is considered to be the first algorithm to effectively solve general polynomial inequality constraints. Several improvements were introduced across the years to reduce the high time complexity of the CAD algorithm [52, 68, 16]. Although the CAD algorithm is sound and complete, it scales poorly with the number of polynomial constraints and their order. Other techniques to solve general polynomial inequality constraints include the use of transformations and approximations to scale the computations. For instance, the authors in [70] incorporated Bernstein polynomials in the Prototype Verification System (PVS) theorem prover; these developments are publicly available in the NASA PVS Library. The library uses the range enclosure propriety of Bernstein polynomials to solve quantified polynomial inequalities. However, the library is not complete for non-strict inequalities [70] and is not practical for higher dimensional polynomials.

Another line of work that is related to our work is the use of machine learning to solve combinatorial problems [48, 82]. In particular, the authors in [48] proposed a graph convolutional neural network (GCNN) to learn heuristics that can accelerate mixed-integer linear programming (MILP) solvers.

Similarly, the NeuroSAT solver [82] uses a message-passing neural network (MPNN) to solve Boolean SAT problems. The authors of [82] showed that NeuroSAT generalizes to novel distributions after training only on random SAT problems. Nevertheless, NeuroSAT is not competitive with state-of-art SAT solvers and it does not have a correctness guarantee.

2.2 Problem Formulation

2.2.1 Notation:

We use the symbols $\mathbb{N}$ and $\mathbb{R}$ to denote the set of natural and real numbers, respectively. We denote by $x = \big(x_1, x_2, \cdots, x_n\big) \in \mathbb{R}^n$ the vector of real-valued variables, where $x_i \in \mathbb{R}$. We denote by $I_n(\underline{d}, \overline{d}) = \big[\underline{d}_1, \overline{d}_1\big] \times \cdots \times \big[\underline{d}_n, \overline{d}_n\big] \subset \mathbb{R}^n$ the n-dimensional hyperrectangle where

$\underline{d} = (\underline{d}_1, \cdots, \underline{d}_n)$ and $\overline{d} = \big(\overline{d}_1, \cdots, \overline{d}_n\big)$ are the lower and upper bounds of the hyperrectangle, respectively. For a real-valued vector $x = \big(x_1, x_2, \cdots, x_n\big) \in \mathbb{R}^n$ and an index-vector $K = (k_1, \cdots, k_n) \in \mathbb{N}^n$, we denote by $x^K \in \mathbb{R}$ the scalar $x^K = x_1^{k_1} \cdots x_n^{k_n}$. Given two multi-indices $K = (k_1, \cdots, k_n) \in \mathbb{N}^n$ and $L = (l_1, \cdots, l_n) \in \mathbb{N}^n$, we use the following notation throughout this paper: $K + L = (k_1 + l_1, \cdots, k_n + l_n)$, $\binom{L}{K} = \binom{l_1}{k_1} \cdots \binom{l_n}{k_n}$, and $\sum\limits_{K \leq L} = \sum\limits_{k_1 \leq l_1} \cdots \sum\limits_{k_n \leq l_n}$. A real-valued multivariate polynomial $p : \mathbb{R}^n \to \mathbb{R}$ is defined as:

$$p(x_1, \ldots, x_n) \;=\; \sum_{k_1=0}^{l_1} \sum_{k_2=0}^{l_2} \ldots \sum_{k_n=0}^{l_n} a_{(k_1,\ldots,k_n)} x_1^{k_1} x_2^{k_2} \ldots x_n^{k_n} \;=\; \sum_{K \leq L} a_K x^K,$$

where $L = (l_1, l_2, \ldots, l_n)$ is the maximum degree of x_i for all $i = 1, \ldots, n$. We denote by $a_p = (a_{(0,0,\ldots,0)}, \ldots, a_{(l_1,l_2,\ldots,l_n)})$ the vector of all the coefficients of polynomial p. We denote the space of multivariate polynomials with coefficients in $\mathbb{R}$ by $\mathbb{R}[x_1, x_2, \cdots, x_n]$. Given a real-valued function $f : \mathbb{R}^n \to \mathbb{R}$, we denote by $L_0^-(f)$ and $L_0^+(f)$ the zero sublevel and zero superlevel sets of f, i.e.,:

$$L_0^-(f) = \{x \in \mathbb{R}^n \big| f\big(x\big) \leq 0\}, \qquad L_0^+(f) = \{x \in \mathbb{R}^n \big| f\big(x\big) \geq 0\}.$$

Finally, a function $f : \mathbb{R}^n \to \mathbb{R}^m$ is called Lipschitz continuous if there exists a positive real constant $\omega_f \geq 0$ such that, for all $x_1 \in \mathbb{R}^n$ and $x_2 \in \mathbb{R}^n$, the following holds:

$$\|f(x_1) - f(x_2)\| \leq \omega_f \|x_1 - x_2\|$$

.

2.2.2 Main Problem:

In this chapter, we focus on two problems namely (Problem 1.1) the feasibility problems that involve multiple *polynomial inequality constraints* with input ranges confined within closed hyperrectangles (it was introduced in chapter 1) and (Problem 2.1) the constrained optimization problem which aims to maximize (or minimize) a polynomial objective function subject to other polynomial inequality constraints and input range constraints.

Given a polynomial objective function $p(x) \in \mathbb{R}[x_1, x_2, \cdots, x_n]$, we define the optimization problem as:

Problem 2.1.

$$\begin{aligned}
&\min_{x \in I_n(\underline{d}, \overline{d})} \; p(x) \quad [\boldsymbol{or} \max_{x \in I_n(\underline{d}, \overline{d})} \; p(x)] \\
\textit{subject to:} \quad & p_1(x_1, \cdots, x_n) \; \leq \; 0, \\
& \vdots \\
& p_m(x_1, \cdots, x_n) \; \leq \; 0
\end{aligned}$$

2.3 Convex Abstraction Refinement: Benefits and Drawbacks

In this chapter, we overview our previously reported framework for using convex abstraction refinement in chapter 1 along with some drawbacks that motivate the need for the proposed framework.

2.3.1 Overview of Convex Abstraction Refinement

Sound and complete algorithms that solve Problem 1.1 are known to be doubly exponential in n with a total running time that it is bounded by $\left(m \, \overline{deg}\right)^{2^n}$ [31], where $\overline{deg}$ is the maximum degree among the polynomials $p_1, \ldots, p_m$. Since the complexity of the problem grows exponentially, it is useful to remove (or prune) subsets of the search space in which the solution is guaranteed not to exist. Since Problem 1.1 asks for an x in $\mathbb{R}^n$ for which all the polynomials are negative, a solution does not exist in subsets of $\mathbb{R}^n$ at which one of the polynomials p_i is always positive (i.e., $L_0^+(p_i)$). In the same way, finding regions of the input space for which some of the polynomials are negative $L_0^-(p_i)$ helps with finding the solution faster.

To find subsets of $L_0^+(p_i)$ and $L_0^-(p_i)$ efficiently, the use of "convex abstractions" of the polynomials was proposed in chapter 1. Starting from a polynomial $p_i(x) \in \mathbb{R}[x]$ and a hyperrectangle $I_n \subset \mathbb{R}^n$, the framework in [106] computes two quadratic polynomials $O_j^{p_i}$ and $U_j^{p_i}$ such that:

$$U_j^{p_i}(x) \leq p(x) \leq O_j^{p_i}(x), \qquad \forall x \in I_n, \tag{2.1}$$

where O and U stands for Over-approximate and Under-approximate quadratic polynomials, respectively, and the subscript j in $O_j^{p_i}(x)$ and $U_j^{p_i}(x)$ encodes the iteration index of the abstraction refinement process. It is easy to notice that the zero superlevel set of $U_j^{p_i}(x)$ is a subset of $L_0^+(p_i)$, i.e., $L_0^+(U_j^{p_i}) \subseteq L_0^+(p_i)$. Similarly, the zero sublevel set of $O_j^{p_i}(x)$ is a subset of $L_0^-(p_i)$, i.e., $L_0^-(O_j^{p_i}) \subseteq L_0^-(p_i)$. Moreover, being convex polynomials, identifying the zero superlevel sets and zero sublevel sets of $O_j^{p_i}(x)$ and $U_j^{p_i}(x)$ can be computed efficiently using convex programming tools. By iteratively refining these upper and lower convex approximations, the framework in chapter 1 was able to rapidly prune the search space until regions with relatively small volumes are identified, at which sound and complete tools such as Z3 8.9 and Yices 2.6 (which are based on the Cylindrical Algebraic Decomposition algorithm) are used to search these small regions, efficiently, to find a solution. It is important to notice that these solvers (especially Yices) are optimized for the cases when the search space is a bounded hyperrectangle.

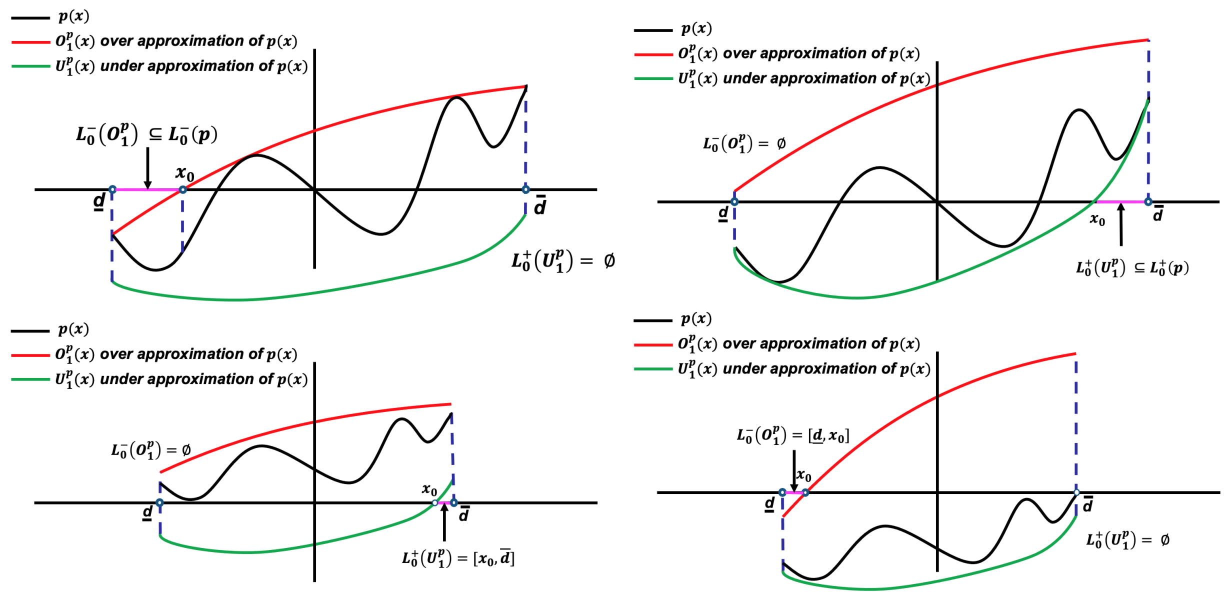


Figure 2.1: Exemplary cases where abstracting higher order polynomial (black curves) using convex approximations fails to provide helpful information: **Top-Left:** under-approximation (green curve) is entirely negative and hence fails to identify any subsets of $L_0^+(p)$. **Top-Right:** over-approximation (red curve) is entirely positive and hence fails to identify subsets of $L_0^-(p)$. **Bottom:** under/over approximations failed to identify polynomials that are consistently positive (left) or negative (right).

2.3.2 Drawbacks of Convex Abstraction Refinement

Although the prescribed convex abstraction refinement process was shown to provide several orders of magnitude speedup compared to the state-of-the-art [106], it adds unnecessary overhead in certain situations. In particular, and as shown in Figure 2.1, the quadratic abstractions $O_j^{p_i}(x)$ and $U_j^{p_i}(x)$ may fail to identify meaningful subsets of $L_0^-(p_i)$ and $L_0^+(p_i)$. One needs to split the input region to tighten the over-/under-approximation in such cases. Indeed, applying the convex abstraction refinement process, above, may lead to several unnecessary over-approximations or under-approximations until a tight one that prunes the search space is found. These drawbacks call for a methodology that is capable of:

1. Guiding the abstraction refinement process: To reduce the number of unnecessary computations of over/under approximations, one needs a heuristic that guides the

convex abstraction refinement process. In particular, such a heuristic needs to consider the properties of the polynomials and the input region to estimate the volume of the sets that the convex under/over-approximation will identify.

2. Alternative Abstraction: As shown in Figure 2.1 (bottom), abstracting high-order polynomials using convex ones may fail to identify easy cases when the polynomial is strictly positive or negative. Therefore, it is beneficial to use alternative ways to abstract high-order polynomials that can augment the convex abstractions.

Designing a strategy that addresses the two requirements above is the main topic for the following two sections.

2.4 Neural Network Guided Convex Abstraction Refinement

In this section, we are interested in designing and training a Neural Network (NN) that can be used to guide the abstraction refinement process. Such NN can be used as an oracle by the solver to estimate the volume of the zero super/sub-level sets (for each polynomial) within a given region $I_n(\underline{d}, \overline{d})$ and select the best approximation strategy out of three possibilities namely: (i) apply convex under-approximation, (ii) apply convex over-approximation, and (iii) split the region to allow for finer approximations in the subsequent iterations of the solver. In this section, we aim to develop a scientific methodology that can guide the design of such NN.

2.4.1 On the relation between the NN architecture and the characteristics of the polynomials:

In this subsection, we aim to understand how the properties of the polynomials affect the design of the NN. We start by reviewing the following result from the machine learning literature:

Theorem 2.1 (Theorem 1.1 [83])**.** *There exists a Rectifier Linear Unit (ReLU)-based neural network ϕ that can estimate a continuous function f such that the estimation error is bounded by:*

$$||\phi - f|| \leq \omega_f \sqrt{d}\, O(N^{-2/d} L^{-2/d})$$

where N, L, d are the neural network depth, the neural network width, and the number of neural network inputs, respectively, and ω_f is the Lipschitz constant of the function f. Moreover, this bound is nearly tight.

The above result can be interpreted as follows. The depth N and width L of a neural network depend on the rate of change of the underlying function (captured by its Lipschitz constant ω_f). That is, if we use a NN to estimate a function with a high ω_f, then one needs to increase the depth N and width L of the NN to achieve an acceptable estimation error.

Now we aim to connect the result above with the characteristics of the polynomials. To that end, we recall the definition of "condition numbers" of a polynomial [34]:

Definition 2.1. *Given a polynomial $p(x) = \sum\limits_{K} a_K x^K$ and a root x_0 of p, the quantity $C_{a_p}(x_0)$ is called the condition number for the root x_0. The condition number characterizes the sensitivity of the root x_0 to a perturbation of the coefficients a_p. That is, if we allow a random perturbation of a fixed relative magnitude $\epsilon = |\frac{\delta a_K}{a_K}|$ in each coefficient a_K in a_p, then the magnitude of the maximum displacement δx_0 of a root x_0 is bounded as: $|\delta x_0| \leq C_{a_p}(x_0)\, \epsilon$. For a polynomial with multiple roots, then we define the condition number of the polynomial*

$\overline{C}_{a_p}$ as the largest $C_{a_p}(x_0)$ among all roots, i.e., $\overline{C}_{a_p} = sup_{x_0 \in \{x | p(x)=0\}} C_{a_p}(x_0)$.

We are now ready to present our first theoretical result that connects the condition number of polynomials to the NN architecture. As stated before, we are interested in designing an NN that can estimate the zero sub/super level volume set within a given region. We show that the larger the condition number, the larger the neural network depth and width, as captured by the following result.

Theorem 2.2. *Given a polynomial p with coefficients a_p, a region $I_n(\underline{d}, \overline{d})$, and an estimate's quality of the volume of zero sub/super level sets l. There exists a neural network $NN(a_p, I_n)$ that estimates the volume of zero sub/super level sets from the polynomial coefficients a_p. The Lipschitz constant of this $NN(a_p, I_n)$, denoted by ω_{NN} is bounded by $\mathcal{O}(n_r \overline{n}_r \overline{C}_{a_p})$ where $\overline{C}_{a_p}$ is the condition number of the polynomial p, $\overline{n}_r = max(l^{\frac{1}{n}}, n_r)$ and n_r is the number of roots of the polynomial p.*

To prove the result, we will proceed with an existential argument. We will show that a NN that matches the properties above exists without constructing such a NN. As shown in Figure 2.2, the neural network $NN(a_p, I_n)$ consists of multiple sub-neural networks. In particular, the first sub-neural network $NN_{a_p \to X_0}$ computes all the roots $X_0 = (x_0^1, \ldots, x_0^{n_r})$ of the polynomial (where n_r is the number of roots) from the coefficients a_p, i.e.:

$$X_0 = NN_{a_p \to X_0}(a_p). \tag{2.2}$$

Note that $NN_{a_p \to X_0}$ does not depend on the region I_n and hence the roots X_0 may not lie inside the region I_n. Moreover, Theorem 2.2 asks for a NN that estimates the volume of the zero sub/super level sets and not the location of the roots. To that end, our strategy is to split the region I_n into sub-regions of fixed volume and check if a root lies within each of these sub-regions. If a sub-region does not have a root (i.e., there is no zero crossing inside this sub-region) and the evaluation of the polynomial at any point in this region turns to be positive, then this sub-region belongs to the super level set of p and similarly for the sublevel set of p. By counting the number of the sub-regions with no zero crossings and multiplying this count by the volume of these sub-regions, we can provide an estimate of the sub/super level sets. Such a process can be performed using the following three sub-neural networks:

- The sub-neural network $NN_{I_n \to I_n^i}$ splits the region I_n into l sub-regions $I_n^1, \ldots, I_n^l$ and return the bounds of the ith sub-region, i.e.:

$$(\underline{d}^i, \overline{d}^i) = NN_{I_n \to I_n^i}(I_n), \qquad i \in \{1, \ldots, l\}. \tag{2.3}$$

- The sub-neural network $NN_{X_0 \to ZC_i}$ checks the location of the roots $(x_0^1, \ldots, x_0^{n_r})$ and returns a binary indicator variable ZC_i that indicates whether a zero-crossing takes place within the ith sub-region or whether the polynomial is always positive/negative within the ith sub-region, i.e.:

$$ZC_i(a_p, I_n^i) = NN_{X_0 \to ZC_i}\left(NN_{a_p \to X_0}(a_p), NN_{I_n \to I_n^i}(I_n)\right). \tag{2.4}$$

- The final output $NN(a_p, I_n)$ is computed using the sub-neural network $NN_{ZC \to L^+/L^-}$ which counts the number of regions that has no zero-crossing (using the indicators $ZC_1, \ldots, ZC_l$) and compute the estimate of the zero sub/super level sets, i.e.:

$$NN(a_p, I_n) = NN_{ZC \to L^+/L^-}\left(ZC_1(a_p, I_n^1), \ldots, ZC_l(a_p, I_n^l)\right). \tag{2.5}$$

The Lipschitz constants of these sub-neural networks are captured by the following four propositions whose proof can be found in the appendix.

Proposition 2.1. *Consider the sub-neural network $NN_{a_p \to X_0}(a_p)$ defined in* (2.2). *The Lipschitz constant of $NN_{a_p \to X_0}(a_p)$ is bounded by $\mathcal{O}(n_r \overline{C}_{a_p})$.*

Proof. Note that $NN_{a_p \to X_0}(a_p)$ is a vector-valued function which returns the roots $X_0 = (x_0^1, x_0^2, \cdots, x_0^{n_r})$ of the polynomial p. Therefore, to upper bound the Lipschitz constant of $NN_{a_p \to X_0}(a_p)$, we will start by upper bounding the Lipschitz constant of its components functions $NN^j_{a_p \to x_0^j}(a_p)$, $1 \leq j \leq n_r$. To that end, consider two polynomials with coefficients a_p and a'_p such that $\left\| a_p - a'_p \right\| \leq \epsilon$. Therefore:

$$\left\| NN^j_{a_p \to x_0^j}(a_p) - NN^j_{a_p \to x_0^j}(a'_p) \right\|_2 = \left\| x_0^j(a_p) - x_0^j(a'_p) \right\|_2 \leq C_{a_p}(x_0^j) \left\| a_p - a'_p \right\| \leq \overline{C}_{a_p} \epsilon \tag{2.6}$$

where $x_0^j(a_p)$ and $x_0^j(a'_p)$ are the location of the jth root for the polynomials with coefficients a_p and a'_p, respectively. The last two inequalities follow from the definition of the condition number (Definition 2.1). Now,

$$\left\| NN_{a_p \to X_0}(a_p) - NN_{a_p \to X_0}(a'_p) \right\|_2 = \sqrt{\sum_{j=1}^{n_r} \left\| NN^j_{a_p \to x_0^j}(a_p) - NN^j_{a_p \to x_0^j}(a'_p) \right\|_2^2} \tag{2.7}$$

$$\leq \sqrt{n_r \overline{C}^2_{a_p} \epsilon^2} = \sqrt{n_r} \overline{C}_{a_p} \epsilon. \tag{2.8}$$

From which we conclude that the Lipschitz constant of $NN_{a_p \to X_0}(a_p)$ is bounded by $\sqrt{n_r} \overline{C}_{a_p} = \mathcal{O}(n_r \overline{C}_{a_p})$. □

Proposition 2.2. *Consider the sub-neural network $NN_{I_n \to I_n^i}$ defined in* (2.3)*. The Lipschitz constant of $NN_{I_n \to I_n^i}$ is bounded by $\mathcal{O}(l^{\frac{1}{n}})$.*

Proof. We assume that the number of sub-regions l is fixed for each dimension n, and $l^{\frac{1}{n}} = k \in \mathbb{N}$. Partitioning the input space into l sub-regions occurs by dividing the interval for each dimension into k sub-intervals. Without loss of generality, the sub-neural network

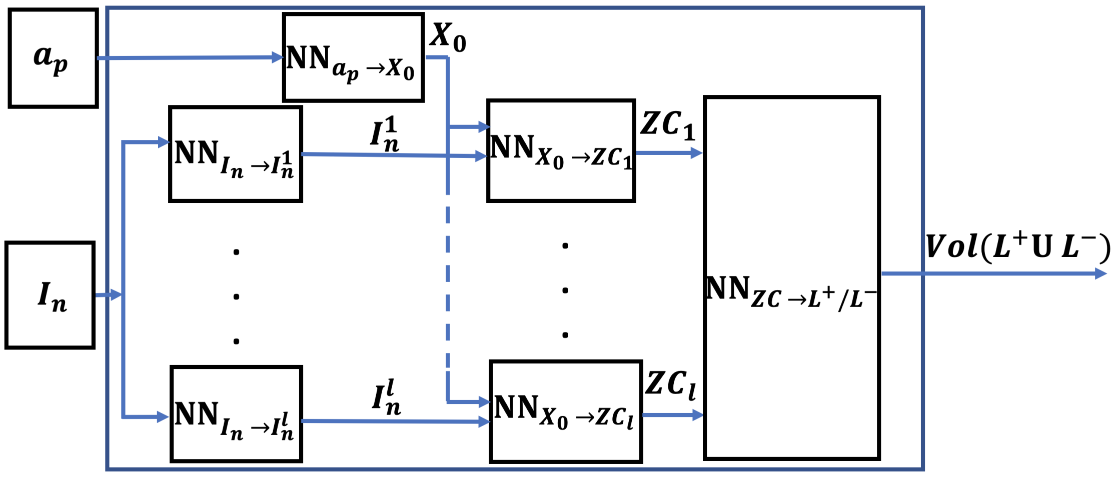

a_p
I_n
$\mathrm{NN}_{a_p \to X_0}$
X_0
$\mathrm{NN}_{I_n \to I_n^1}$
I_n^1
$\mathrm{NN}_{X_0 \to ZC_1}$
ZC_1
$\mathrm{NN}_{I_n \to I_n^l}$
I_n^l
$\mathrm{NN}_{X_0 \to ZC_l}$
ZC_l
$\mathrm{NN}_{ZC \to L^+/L^-}$
$Vol(L^+ \cup L^-)$

Figure 2.2: The architecture of the neural network $NN(a_p, I_n)$ used to prove the Theorem 2.2.

$NN_{I_n \to I_n^i}(I_n)$ for $n = 1$ can be defined as:

$$(\underline{d}^i, \overline{d}^i) = NN_{I_n \to I_n^i}(I_n) = NN_{I_n \to I_n^i}(\underline{d}, \overline{d}) = \left(\underline{d} + \frac{i}{k}\left(\overline{d} - \underline{d}\right), \overline{d} + \frac{i+1}{k}\left(\overline{d} - \underline{d}\right)\right)$$
$$= \begin{bmatrix} 1 + i/k & -i/k \\ -(i+1)/k & 1 + (i+1)/k \end{bmatrix} \begin{bmatrix} \underline{d} \\ \overline{d} \end{bmatrix} \tag{2.9}$$

A generalization to a higher dimension is straightforward by replacing i with a multi-index in each dimension. Note that $NN_{I_n \to I_n^i}(I_n)$ is a multivariate linear function in its inputs and hence its Lipschitz constant can be computed as the largest singular value. Indeed, the linear function depends only on the constant k (which depends on the constant l and the dimension n) from which we conclude that the Lipschitz constant of $NN_{I_n \to I_n^i}$ is $\mathcal{O}(l^{\frac{1}{n}})$. □

Proposition 2.3. *Consider the sub-neural network $NN_{X_0 \to ZC_i}(X_0, I_n^i)$ defined in (2.4). The Lipschitz constant of $NN_{X_0 \to ZC_i}(X_0, I_n^i)$ is bounded by $\mathcal{O}(n_r)$.*

Proof. It follows from equations (2.2)-(2.4) that the sub-neural network $NN_{X_0 \to ZC_i}$ can be written as:

$$\begin{aligned} ZC_i(a_p, I_n^i) &= NN_{X_0 \to ZC_i}\left(NN_{a_p \to X_0}(a_p), NN_{I_n \to I_n^i}(I_n)\right) \\ &= NN_{X_0 \to ZC_i}\left((x_0^1, \ldots, x_0^{n_r}), (\underline{d}^i, \overline{d}^i)\right). \end{aligned} \tag{2.10}$$

The indicator variable ZC_i should be set to zero whenever all the roots x_0^j lies outside the hyperrectangle $I_n^i(\underline{d}^i, \overline{d}^i)$. First note that a root x_0^j lies outside $I_n^i(\underline{d}^i, \overline{d}^i)$ if and only if the following condition holds:

$$x_0^j \notin I_n^i(\underline{d}^i, \overline{d}^i) \quad \iff \quad \sum_{k=1}^{k=n} \left| x_{0,k}^j - \underline{d}_k^i \right| + \left| x_{0,k}^j - \overline{d}_k^i \right| - \sum_{k=1}^{n} \left(\overline{d}_k^i - \underline{d}_k^i \right) > 0 \tag{2.11}$$

where $x_{0,k}^j$, $\underline{d}_k^i$, and $\overline{d}_k^i$ are the kth element in the vectors $x_0^j, \underline{d}^i$ and $\overline{d}^i$, respectively. Hence, the indicator variable ZC_i should be set to zero whenever the following conditions hold:

$$\begin{aligned} ZC_i(a_p, I_n^i) = 0 \quad \iff & \\ & \max_{j \in \{1,\ldots,n_r\}} \left(\sum_{k=1}^{k=n} \left| x_{0,k}^j - \underline{d}_k^i \right| + \left| x_{0,k}^j - \overline{d}_k^i \right| - \sum_{k=1}^{n} \left(\overline{d}_k^i - \underline{d}_k^i \right) \right) = 0 \end{aligned} \tag{2.12}$$

Before we compute the Lipschitz constant of $NN_{X_0 \to ZC_i}$ in Equation (2.12), we recall the following identities. Consider two functions $f(x)$ and $g(x)$ with Lipschitz constants L_f and L_g, respectively. Then:

- The Lipschitz constant of $\max(f(x), g(x))$ is bounded by $L_f + L_g$.
- The Lipschitz constant of $f(x) + g(x)$ is bounded by $\max(L_f, L_g)$.

Now notice that $|x_{0,k}^j - \underline{d}_k^i| = max(x_{0,k}^j - \underline{d}_k^i, 0) + max(-x_{0,k}^j + \underline{d}_k^i, 0)$. Applying the identities above along with the fact that the Lipschitz constant of $x_{0,k}^j - \underline{d}_k^i$ is $\mathcal{O}(1)$, we conclude that the Lipschitz constant of $|x^j_{0,k} - \underline{d}^i_k|$ is $\mathcal{O}(1)$. Hence, the Lipschitz constant of $\sum_{k=1}^{k=n} \left| x^j_{0,k} - \underline{d}^i_k \right| + \left| x^j_{0,k} - \overline{d}^i_k \right| - \sum_{k=1}^{n} \left(\overline{d}^i_k - \underline{d}^i_k \right)$ is also $\mathcal{O}(1)$. Finally, the Lipschitz constant of the right hand side of Equation (2.12) is $\mathcal{O}(n_r)$. We conclude our proof by noticing that all the operators in Equation (2.12)—namely the absolute value, the max operator, summation, and checking the final value against a constant—can be implemented exactly using ReLU neural networks [38] and hence the neural network $NN_{X_0 \to ZC_i}$ will also have a Lipschitz constant equal to $\mathcal{O}(n_r)$. □

Proposition 2.4. *Consider the sub-neural network $NN_{ZC \to L^+/L^-}$ defined in* (2.5). *The Lipschitz constant of $NN_{ZC \to L^+/L^-}$ is bounded by $\mathcal{O}(1)$.*

Proof. This result follows directly by noticing that $NN_{ZC \to L^+/L^-}$ can be computed as a linear function:

$$NN_{ZC \to L^+/L^-}(ZC_1, \ldots, ZC_l) = v \sum_{i=1}^{l} (1 - ZC_i) \tag{2.13}$$

where v is a constant that depends on the volume of the hyperrecatngle I_n. Since l is a constant, we conclude the result. □

Proof of Theorem 2.2. Consider the NN shown in Figure 2.2 and defined using equations (2.2)-(2.5). To bound the Lipschitz constant of $NN(a_p, I_n)$, we consider two sets of inputs (a_p, I_n)

and (a'_p, I'_n) as follows:

$$\begin{aligned}
&\left\|NN(a'_p, I'_n) - NN(a_p, I_n)\right\|_2 \\
&= \Big\|NN_{ZC\to L^+/L^-}\left(ZC_1(a'_p, I'^1_n), \ldots, ZC_l(a'_p, I'^l_n)\right) \\
&\qquad\qquad -NN_{ZC\to L^+/L^-}\left(ZC_1(a_p, I^1_n), \ldots, ZC_l(a_p, I^l_n)\right)\Big\|_2, && (2.14)\\
&\leq \mathcal{O}(1)\left\|\left(ZC_1(a'_p, I'^1_n) - ZC_1(a_p, I^1_n), \cdots, ZC_l(a'_p, I'^l_n) - ZC_l(a_p, I^l_n)\right)\right\|_2, && (2.15)\\
&= \mathcal{O}(1)\Bigg(\sum_{i=1}^{l}\left\|ZC_i(a'_p, I'^i_n) - ZC_i(a_p, I^i_n)\right\|_2^2\Bigg)^{\frac{1}{2}}. && (2.16)
\end{aligned}$$

where (2.15) follows from Proposition 2.4. Now, we upper bound $\left\|ZC_i(a'_p, I'^i_n) - ZC_i(a_p, I^i_n)\right\|_2$ as follows:

$$\begin{aligned}
&\left\|ZC_i(a'_p, I'^i_n) - ZC_i(a_p, I^i_n)\right\|_2^2 \\
&= \Big\|NN_{X_0\to ZC_i}\left(NN_{a_p\to X_0}(a'_p), NN_{I_n\to I^i_n}(I'_n)\right) \\
&\qquad\qquad -NN_{X_0\to ZC_i}\left(NN_{a_p\to X_0}(a_p), NN_{I_n\to I^i_n}(I_n)\right)\Big\|_2^2 && (2.17)\\
&\leq \mathcal{O}(n_r)\left\|\left(NN_{a_p\to X_0}(a'_p) - NN_{a_p\to X_0}(a_p), NN_{I_n\to I^i_n}(I'_n) - NN_{I_n\to I^i_n}(I_n)\right)\right\|_2^2, && (2.18)\\
&= \mathcal{O}(n_r)\left(\left\|\left(NN_{a_p\to X_0}(a'_p) - NN_{a_p\to X_0}(a_p)\right)\right\|_2^2 + \left\|NN_{I_n\to I^i_n}(I'_n) - NN_{I_n\to I^i_n}(I_n)\right\|_2^2\right), \\
&\leq \mathcal{O}(n_r)\left(\mathcal{O}(n_r\overline{C}_{a_p})\left\|a'_p - a_p\right\|_2^2 + \mathcal{O}(l^{\frac{1}{n}})\left\|I'_n - I_n\right\|_2^2\right) && (2.19)\\
&= \mathcal{O}(n_r\overline{n}_r\overline{C}_{a_p})\left\|(a'_p, I'_n) - (a_p, I_n)\right\|_2^2. && (2.20)
\end{aligned}$$

where (2.18) follows from Proposition 2.3; (2.19) follows from Propositions 2.2 and 2.1 along with the definition of $\overline{n}_r = \max(l^{\frac{1}{n}}, n_r)$. Substituting (2.20) in (2.16) and noticing that l is a constant that does not depend on n yields:

$$\left\|NN(a'_p, I'_n) - NN(a_p, I_n)\right\|_2 \leq \mathcal{O}(n_r \overline{n}_r \overline{C}_{a_p}) \left\|(a'_p, I'_n) - (a_p, I_n)\right\|_2, \tag{2.21}$$

from which we conclude that the Lipschitz constant of $NN(a_p, I_n)$ is in the order of $\mathcal{O}(n_r \overline{n}_r \overline{C}_{a_p})$ which concludes the proof of Theorem 2.2. $\square$

According to Theorem 2.1, the depth and width of an NN must be increased to achieve an acceptable estimation error proportional to the NN's Lipschitz constant. Additionally, Theorem 2.2 establishes that the Lipschitz constant of $NN\left(a_p, I_n\right)$ is upper-bounded by a constant dependent on the condition number of the polynomial $\overline{C}_{a_p}$. Consequently, we can deduce from Theorem 2.1 and Theorem 2.2 that higher condition numbers of polynomials necessitate larger network widths and depths for accurate estimation of zero sub/super level set volumes. Notably, the power basis representation (i.e., representing the polynomial as a summation $\sum_{K \leq L} a_K x^K$) has been identified to possess an unstable nature with significantly large condition numbers [34], thereby demanding neural networks with substantial architectural complexities which motivates the need to use other polynomial representations.

2.4.2 Bernstein Polynomials: A Robust Representation of Polynomials

Motivated by the challenge above, we seek a representation of polynomials that is more robust to changes in coefficients, i.e., we seek a representation in which the roots of the polynomial change slowly with changes in the coefficients (and hence smaller condition numbers $\overline{C}_{a_p}$ and a smaller NN to estimate the volume of the sub/super level sets). We start with the following definition.

Definition 2.2. *Let* $p(x) = \sum_{K \le L} a_K x^K \in \mathbb{R}[x_1, \dots, x_n]$ *be a multivariate polynomial over a hyperrectangle* $I_n(\underline{d}, \overline{d})$ *and of a maximal degree* $L = (l_1, \cdots, l_n) \in \mathbb{N}^n$. *The polynomial:*

$$B_{p,L}(x) = \sum_{K \le L} b_{K,L} Ber_{K,L}(x), \tag{2.22}$$

is called the Bernstein polynomial of p, *where* $Ber_{K,L}(x)$ *and* $b_{K,L}$ *are called the Bernstein basis and Bernstein coefficients of* p, *respectively, and are defined as follows:*

$$Ber_{K,L}(x) = \binom{L}{K} x^K (1-x)^{L-K}, \qquad b_{K,L} = \sum_{J=(0,\dots,0)}^{K} \frac{\binom{K}{J}}{\binom{L}{J}} \left(\overline{d} - \underline{d}\right)^J \sum_{I=J}^{L} \binom{I}{J} \underline{d}^{I-J} a_I. \tag{2.23}$$

The Bernstein representation is known to be the most robust representation of polynomials which is captured by the next result [34].

Theorem 2.3 (Theorem [34])**.** *The Bernstein basis is optimally stable, i.e. there exists no other basis with a condition number smaller than the condition number of the Bernstein coefficients* $\overline{C}_{b_p}$, *where* $b_p = (b_{(0,0,\dots,0),L}, \dots, b_{(l_1,l_2,\dots,l_n),L})$ *is the vector of all the Bernstein coefficients of polynomial* p.

Theorems 2.1-2.3 point to the optimal way of designing the targeted neural network. Such a neural network needs to take as input the Bernstein coefficients b_p instead of the power basis coefficients a_p. To validate this conclusion, we report empirical evidence in Table 2.1. In this numerical experiment, we trained two neural networks with the same exact architecture, using the same exact number of data points, and both networks have the same number of inputs. Both neural networks are trained to estimate whether a zero-crossing occurs in a region (recall from our analysis in Theorem 2.2 that the Lipschitz constant of this NN is equal to the condition number of the polynomial). The only difference is that one neural network is trained using power basis coefficients a_p (column 3 of Table 2.1) while the second is trained

Table 2.1: Evaluation of three trained neural networks on three different benchmarks for the different polynomial basis. Each benchmark has 10000 samples. The coefficients of the polynomial within each basis are generated following a uniform distribution given in the table.

Benchmark	Coefficients	Power Basis		Bernstein Basis		Reduced Bernstein Basis	
		Accuracy	Overhead	Accuracy	Overhead	Accuracy	Overhead
1	$\mathcal{U}(-0.1, 0.1)$	46%	0 [s]	91%	0.01 [s]	82%	0.002 [s]
2	$\mathcal{U}(-0.5, 0.5)$	32%	0 [s]	87%	0.03 [s]	79%	0.005 [s]
3	$\mathcal{U}(-1, 1)$	30%	0 [s]	88%	0.04 [s]	80%	0.007 [s]

using Bernstein basis coefficients b_p (column 5 of Table 2.1). The coefficients are randomly generated via a uniform distribution between -0.1 and 0.1, i.e., $\mathcal{U}(-0.1, 0.1)$. We generated 40000 training samples and 10000 validation samples for both bases. We evaluate the trained NN on three different benchmarks for the two bases. Each evaluation benchmark has 10000 samples. The results are summarized in Table 2.1. As it can be seen from Table 2.1, the NN trained with Bernstein coefficients generalizes better than the NN trained with power basis coefficients as reflected by the empirical "Accuracy" during evaluation. This empirical evidence matches our analysis in Theorem 2.2 along with the insights of Theorem 2.1 and Theorem 2.3.

2.5 Taming the Complexity of Computing Bernstein Coefficients

In section 4, we concluded that Bernstein's representation has a smaller condition compared to other representations, which helps build a more efficient NN. Nevertheless, computing this representation adds a significant overhead even by using the most efficient algorithms to calculate these coefficients [85, 77].

For example, computing all the Bernstein coefficients of a 6^{th}-dimensional polynomial with 7^{th} order using Matrix method and Garloff's methods [85, 77] require $1.1e07$ and $7.1e06$

summation and multiplication operations [77].

To exacerbate the problem, the Bernstein coefficients depend on the region I_n and need to be recomputed in every iteration of the abstraction refinement process. Reducing such overhead is the main focus of this section.

2.5.1 Range Enclosure Property of Bernstein polynomials

Given a multivariate polynomial $p(x)$ that is defined over the n-dimensional box $I_n(\underline{d}, \overline{d})$, we can bound the range of $p(x)$ over $I_n(\underline{d}, \overline{d})$ using the range enclosure property of Bernstein polynomials as follows:

Theorem 2.4 (Theorem 2 [44]). *Let p be a multivariate polynomial of degree L over the n-dimensional box $I_n(\underline{d}, \overline{d})$ with Bernstein coefficients $b_{K,L}$, $0 \leq K \leq L$. Then, for all $x \in I_n$, the following inequality holds:*

$$\min_{K \leq L} \ b_{K,L} \leq p(x) \leq \max_{K \leq L} \ b_{K,L}. \tag{2.24}$$

The traditional approach to computing the range enclosure of p is to compute all the Bernstein coefficients of p to determine their minimum and maximum [45, 46, 107]. However, computing all the coefficients has a complexity of $\mathcal{O}\left((l_{max}+1)^n\right)$, where $l_{max} = \max\limits_{1 \leq i \leq n} l_i$, which increases exponentially with the dimension n. Luckily, the Bernstein coefficients enjoy monotonicity properties, whenever the region $I_n(\underline{d}, \overline{d})$ is restricted to be an orthant (i.e., the sign of x_i does not change within $I_n(\underline{d}, \overline{d})$, for each $i \in \{1, \ldots, n\}$) [85]. Using such monotonicity properties, one can compute the minimum and maximum Bernstein coefficients (denoted by $\underline{B}_{p,L}$, $\overline{B}_{p,L}$) with a time complexity of $\mathcal{O}\left(2\left(l_{max}+1\right)^2\right)$ which does not depend on the dimension n.

2.5.2 Zero Crossing Estimation using only a few Bernstein Coefficients

Now we discuss how to use the range enclosure property above to reduce the number of computed Bernstein coefficients. First, we note that the zero crossing of a polynomial p in a given input region I_n depends on its estimate range given by $\underline{B}_{p,L}$ and $\overline{B}_{p,L}$. More specifically, if $\underline{B}_{p,L} > 0$ ($\overline{B}_{p,L} < 0$), then the entire polynomial is positive (negative), which means that there is no zero-crossing. If $\underline{B}_{p,L}$ and $\overline{B}_{p,L}$ have different signs, and because of the estimation error of these bounds, the polynomial p may still be positive, negative, or have a zero crossing in the region. In this case, we need additional information such as the bounds of the gradient of the polynomial p within the input region, that are given by $\underline{B}_{\nabla p,L}$ and $\overline{B}_{\nabla p,L}$ (which can be computed efficiently thanks to the fact that gradients of polynomials are polynomials themselves). Such additional information about the worst-case gradient of the polynomial leads to a natural estimate of whether a zero crossing occurs in a region.

Due to space constraints, we omit the analysis of bounding the estimation error introduced by relying only on the maximum and minimum of the polynomial $\underline{B}_{p,L}$ and $\overline{B}_{p,L}$ along with the maximum and minimum of the gradient $\underline{B}_{\nabla p,L}$ and $\overline{B}_{\nabla p,L}$. Instead, we support our claim using the empirical evidence shown in Table 2.1. Using the same benchmarks used in Section 4.2, we train a third neural network that takes as input only the four inputs $\underline{B}_{p,L}, \overline{B}_{p,L}, \underline{B}_{\nabla p,L}, \overline{B}_{\nabla p,L}$ and compare its generalization performance (column 7 of Table 2.1). As shown in the table, the third neural network sacrifices some accuracy compared to the ones that use all Bernstein coefficients. But on the other side, it reduces the overhead to compute the Bernstein coefficients by order of magnitude as can be seen by comparing the "execution overhead" reported in columns 4, 6, and 8 for the power basis, the Bernstein basis, and the reduced Bernstein basis, respectively.

2.5.3 Search Space pruning using Bernstein Coefficients

The range enclosure property and the discussion above open the door for a natural solution of the "alternative abstraction" problem mentioned in Section 3.2. The maximum and minimum Bernstein coefficients can be used as an abstraction (in addition to convex upper and lower bounds) of high-order polynomials. Such abstractions can be refined with every iteration of the solver. They can be used to identify portions of the search space for which one of the polynomials is guaranteed to be positive (and hence a solution does not exist). More details about integrating this abstraction and the convex abstraction are given in the implementation section below.

2.6 Algorithm Architecture and Implementation Details

In this section, we describe the implementation details of our solver PolyARBerNN. As a pre-processing step, the tool divides the input region I_n into several regions such that each one is an orthant. This allows the tool to process each orthant in parallel or sequentially. The tool keeps track of all regions for which the sign of a polynomial is not fixed. These regions are called ambiguous regions, and they are stored in a list called *Ambig*. As long as the volume of the regions in this list is larger than a user-defined threshold ϵ, then our tool will continuously use abstractions to identify portions in which one of the polynomials is always positive (and hence removed from the search space) or negative (and hence the tool will give higher priority for this region). The abstraction refinement is iteratively applied in Lines 5-17 of Algorithm 1. In each abstraction refinement step, the tool picks a polynomial p and a region *region* based on several heuristics (Lines 6-7). In lines 8-14, we compute the maximum/minimum Bernstein coefficients followed by checking the sign of the polynomial

Algorithm 5 PolyARBerNN

```
Input: $I_n(\underline{d}, \overline{d}), p_1, p_2, \ldots, p_m, \epsilon,$        Output: $x_{\text{Sol}}$

1: $orthants$ := Partition_Region($I_n$)
2: $Neg$ := {}
3: $Ambig$ := {$orthants$}
4: List_pols := {$p_1, \ldots, p_m$}
5: while Compute_Maximum_Volume($Ambig$) $\geq \epsilon$ do
6:     $p$ := Select_Poly (List_pols, $Neg$)
7:     $region$ := Remove_Ambiguous_Region_From_List ($Ambig$)
8:     $\left(\underline{B}_{p,L}, \overline{B}_{p,L}, \underline{B}_{\nabla p,L}, \overline{B}_{\nabla p,L}\right)$ := Compute_Bern_Coeff($p, region$)
9:     if $\underline{B}_{p,L} > 0$ then
10:        break
11:    else if $\underline{B}_{p,L} < 0$ then
12:        $Neg := Neg \cup (p, L_0^-(p))$
13:        break
14:    end if
15:    (under_approx, over_approx, split) := $NN(\underline{B}_{p,L}, \overline{B}_{p,L}, \underline{B}_{\nabla p,L}, \overline{B}_{\nabla p,L}, region)$
16:    $action$ := Select_best_action(under_approx, over_approx, split)
17:    $L_0^-(p), L_0^+(p), L_0^{+/-}(p)$ := Convex_Abst_Refin_PolyAR ($p, action, region$)
18:    $Ambig := Amibg \cup L_0^{+/-}(p)$
19:    $Neg := Neg \cup (p, L_0^-(p))$
20: end while
21: if is_List_Empty($Ambig$) then
22:    if A negative region in $Neg$ has all the polynomials then
23:        $x_{\text{Sol}}$ := any point in the negative region
24:    else
25:        return  the problem is UNSAT
26:    end if
27: else
28:    $x_{\text{Sol}}$ := CAD_Solver_Parallel ($Ambig, p_1, \ldots, p_m$)
29: end if
```

within this region. Suppose the Bernstein coefficients indicate that the polynomial is always positive in this region. In that case, this provides a guarantee that a solution does not exist in this region (recall that Problem 1 searches for a point where *all* polynomials are negative). Similarly, if the polynomial is always negative, then it will be added to the list of negative regions. For those polynomials for which the Bernstein abstraction failed to identify their signs, we query the trained neural network to estimate the best convex abstraction possible

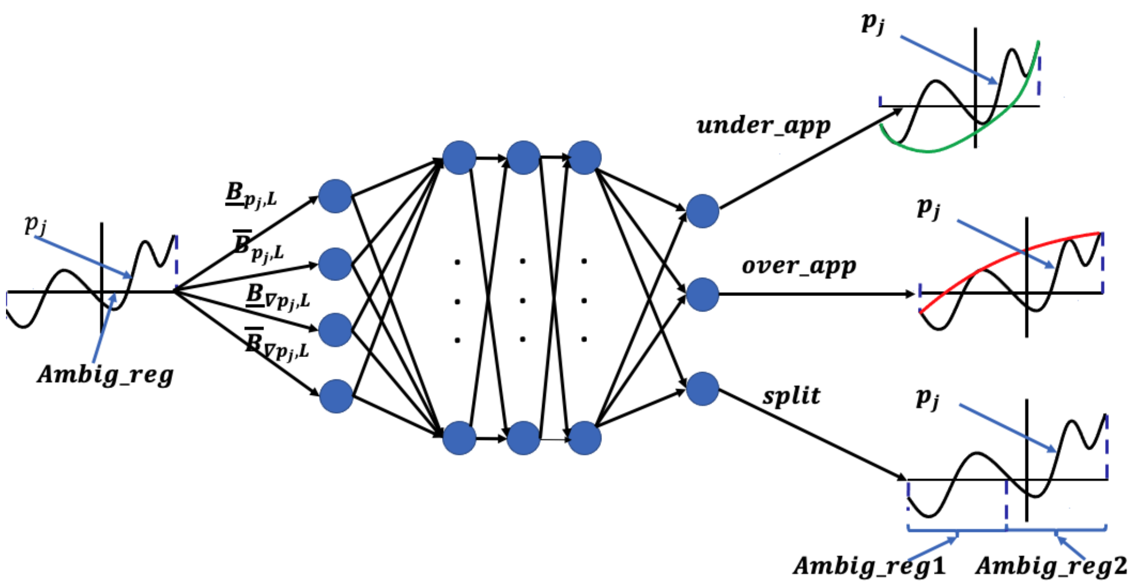


Figure 2.3: The architecture of the trained NN that is used to guide the abstraction refinement process within PolyARBerNN. We used a fully connected NN that contains an input layer with 4 neurons, three hidden layers with 40 neurons each, and one output layer with three neurons. All neurons are ReLU-based except for the output neurons which uses SoftMax non-linearity.

(Lines 15-16). Based on the neural network suggestion, we use the PolyAR tool [106] to compute the convex abstraction (Line 17), which returns portions of this region that are guaranteed to belong to the zero sublevel set $L_0^-(p)$, those who belong to the zero superlevel set $L_0^+(p)$, and those remain ambiguous $L_0^{+/-}(p)$. The process of using Bernstein abstraction and the convex abstraction (which is guided by the trained neural network) continues until all remaining ambiguous regions are smaller than a user-defined threshold ϵ in which case it will be processed in parallel using a sound and complete tool that implements Cylindrical Algebraic Decomposition (CAD) such as Z3 and Yices (line 28 in Algorithm 1).

The neural network itself is trained using randomly generated, quadratic, two-dimensional polynomials where the coefficients follow a uniform distribution between -1 and 1. For each randomly generated polynomial, we used PolyAR to compute the volumes of the $L_0^+(p), L_0^-(p), L_0^{+/-}(p)$ regions. We use a fully connected NN that contains an input layer, three hidden layers, and one output layer (shown in Figure 2.3). The input layer has four neurons, the hidden layers have 40 neurons each, and the output layer has three neurons. We use a dropout of probability 0.5 in the first and second hidden layers to avoid overfitting. We use the ReLU activation function for all the hidden layers and the Softmax activation function for the output layer. We use Adam as an optimizer and cross-entropy as a loss function. Although the neural network is trained on simple quadratic two-dimensional polynomials, we observed it generalizes well to higher-order polynomials with several variables. This will become apparent during the numerical evaluation in which polynomials of different orders and several variables will be used to evaluate the tool.

Correctness Guarantees: We conclude our discussion with the following result which captures the correctness guarantees of the proposed tool:

Theorem 2.5. *The PolyARBerNN solver is sound and complete.*

Proof. This result follows from the fact that search space is pruned using sound abstractions (convex upper bounds or Bernstein-based). The neural network and the convex lower bound polynomials are just used as heuristics to guide the refinement process. Finally, CAD-based algorithms (which are sound and complete) are used to process the portions of the search space which are not pruned by the abstraction refinement. □

2.7 Generalization to polynomial optimization problems:

In this section, we focus on providing a solution to Problem 2.1. Our approach is to turn the optimization problem (Problem 2.1) into a feasibility problem (Problem 1.1). First, we recall that the gradient of p, $\nabla p = [\frac{\partial p}{\partial x_1}, \cdots, \frac{\partial p}{\partial x_n}]$, where $\frac{\partial p}{\partial x_i}$ is the partial derivative of p with respect to x_i, is a vector of n polynomials. The optimal value of p occurs either (i) when the vector of partial derivatives are all equal to zero or (ii) at the boundaries of the input region.

To find the critical points x^* of p where $\nabla p\left(x^*\right) = 0$, we add the n polynomial constraints $\frac{\partial p}{\partial x_i} \leq 0, 1 \leq i \leq n$ and $-\frac{\partial p}{\partial x_i} \leq 0, 1 \leq i \leq n$ to the constraints of the optimization problem. Now, we modify the PolyARBernNN solver to output *all* possible regions in which all the constraints are satisfied. This can be easily computed by taking the intersections within the regions stored in the data structure Neg in Algorithm 5. These regions enjoy the property that *all* points in these regions are critical points of p. In addition, we modify PolyARBernNN to output *all* the remaining ambiguous regions whose volumes are smaller than the user-specified threshold ϵ and for which the CAD-based solvers returned a solution. These regions enjoy the property that *there exists* a point inside these regions which is a critical point. These modifications are captured in (Line 2 of Algorithm 6).

Since the minimum/maximum of p may occur at the boundaries of the region $I_n\left(\underline{d}, \overline{d}\right)$, our solver samples from the boundaries of the region $I_n\left(\underline{d}, \overline{d}\right)$ (Line 4 in Algorithm 6). The solver uses $\delta = 2\sqrt{n}(\epsilon)^{1/n}$ as sampling distance between two successive boundary samples—recall ϵ is a user-defined parameter and was used in Algorithm 1 as a threshold on the refinement process. Next, we evaluate the polynomial p in the obtained samples (line 5 Algorithm 6). Then, we take the minimum and the maximum over the obtained values (Line 7 in Algorithm 6). All the details can be found in Algorithm 6.

We conclude our discussion with the following result which captures the error between the solutions provided by PolyAROpt and the global optima.

Theorem 2.6. *Let p^*_{min} and p^*_{max} be the global optimal points for the solution of Problem 2. The solutions obtained by Algorithm 2, denoted by $\hat{p}_{min}$ and $\hat{p}_{max}$ satisfy the following:*

$$|\hat{p}_{min} - p^*_{min}| \leq \omega_p \delta, \qquad |\hat{p}_{max} - p^*_{max}| \leq \omega_p \delta, \tag{2.25}$$

Algorithm 6 PolyAROpt

```
Input: $I_n(\underline{d}, \overline{d}), p, p_1, p_2, \ldots, p_m, \epsilon$
Output: $\hat{p}_{min}$, $\hat{p}_{max}$

1: $\nabla p = \texttt{Grad\_Poly}(p)$
2: $\hat{reg}_{\text{list}} = \texttt{PolyARBerNN}(I_n(\underline{d}, \overline{d}), \nabla p, p_1, \ldots, p_m, \epsilon)$
3: $\hat{x}_{\text{list}} = \texttt{center}(\hat{reg}_{\text{list}})$
4: $x^{end}_{list} = \texttt{Sample\_boundaries}(I_n(\underline{d}, \overline{d}), \epsilon)$
5: $\hat{p}_{list} = p(\hat{x}_{\text{list}})$;    $p^{end}_{list} = p(x^{end}_{\text{list}})$
6: $p_{list} = \hat{p}_{list} \cup p^{end}_{list}$
7: $\hat{p}_{min} = \min(p_{list})$;    $\hat{p}_{max} = \max(p_{list})$
```

where ω_p is the Lipschitz constant of the polynomial p, $\delta = 2\sqrt{n}(\epsilon)^{1/n}$ is the sampling distance, and $\epsilon > 0$ is a user-defined error.

Proof. Let us denote by x_{list} the input domain points that correspond to p_{list}. Let us denote by $\overline{x}_{min} = \min_{x \in x_{list}} \|x - x^*_{min}\|$ and $\overline{x}_{max} = \min_{x \in x_{list}} \|x - x^*_{max}\|$, the nearest point to the actual optimal points x^*_{min} and x^*_{max}. We note that there are three cases that Algorithm 2 uses to compute the set of critical points, x_{list}:

1. Using the center of the regions in the Neg list
2. Using the center of the regions in the $Ambig$ list
3. Using samples from the boundaries

We proceed by case analysis. **Case 1:** First, we note that *all* the points within the Neg regions satisfy that $\nabla p = 0$ and hence the value of the polynomial takes the same exact value overall the region, hence the value of p at the center $\hat{x}$ of the region is the same at the global optima x^*. **Case 2 and Case 3:** First, we can show that $\overline{x}_{min}$ and $\overline{x}_{max}$ are bounded from the actual optimal points x^*_{min} and x^*_{max} by:

$$\|\overline{x}_{min} - x^*_{min}\| \leq \delta \qquad\qquad \|\overline{x}_{max} - x^*_{max}\| \leq \delta \tag{2.26}$$

where $\delta = 2\sqrt{n}(\epsilon)^{1/n}$. To show that inequalities (2.26) hold we proceed by case analysis. However (2.26) follow directly in Case 2 from the fact that *Ambig* regions have a volume that is smaller than ϵ (Line 7 in Algorithm 1) and hence the distance between any two points within the regions is bounded by $\delta = 2\sqrt{n}(\epsilon)^{1/n}$. Similarly, Case 3 follows from the fact that Algorithm 2 samples from the boundaries with a maximum distance between the samples that is equal to $\delta = 2\sqrt{n}(\epsilon)^{1/n}$. Second, we obtain (2.25) as follows:

$$\hat{p}_{min} \leq p(\overline{x}_{min}) \Rightarrow |\hat{p}_{min} - p^*_{min}| \leq |p(\overline{x}_{min}) - p^*_{min}| \leq \omega_p \delta, \tag{2.27}$$

where 2.27 comes from the definition of $\hat{p}_{min}$, the Lipschitz continuity of polynomial p, and 2.26. The inequality for $|\hat{p}_{max} - p^*_{max}|$ is obtained in a similar manner. $\square$

2.8 Numerical Results - NN Training

In this section, we show the details of training and evaluating the NN used to help PolyARBerNN selecting the best convex abstraction. We evaluate the trained NN on six different benchmarks. The benchmarks are different than the training benchmarks with respect to the input region, the degree of the polynomial, and the number of variables of the polynomial. All the experiments were executed on an Intel Core i7 2.6-GHz processor with 16 GB of memory.

2.8.1 Training data collection and pre-processing

Data collection

To collect the data, we generated random quadratic two-dimensional polynomials:

$$q\left(x_1,\ x_2\right) \ = \ c_1 x_1^2 + c_2 x_2 + c_3 x_1 x_2 + c_4 x_2 + c_5 y + c_6,$$

where the coefficients $c_1, \ldots, c_6$ follow a uniform distribution between -1 and 1.The random generated polynomials are defined over the domain $I_2 = \big[-2, 2\big]^2$. For each randomly generated polynomial, we perform the abstraction refinement on the domain I_2, iteratively. In every iteration, we perform under-approximation, over-approximation of the original polynomial over a selected ambiguous region, and a split of the ambiguous region. Next, we compute the volume of the remaining ambiguous region after each action was implemented.

The labels are a one-hot vector of dimension three where each component represents the action that leads to the maximum reduction in the volume of the ambiguous region, either under-approximation, over-approximation or divide the region into two regions.

We ran the abstraction refinement process on all the generated polynomials to collect the data $\big(\underline{B^i}_{p_j,L}, \overline{B^i}_{p_j,L}, \underline{B^i}_{\nabla p_j,L}, \overline{B^i}_{\nabla p_j,L}\big)$, where i denote the index of the sample. We generate 50000 samples for training, 10000 samples for validation, and 10000 for testing.

Data Normalization

In the literature of NN [61], it is important to normalize the data when the data vary across a wide range of values. This normalization leads to faster training and improves the generalization performance of the NN [61]. Therefore, we normalize all the input data to a zero mean and unit variance by adopting a simple affine transformation data_sample $\leftarrow$

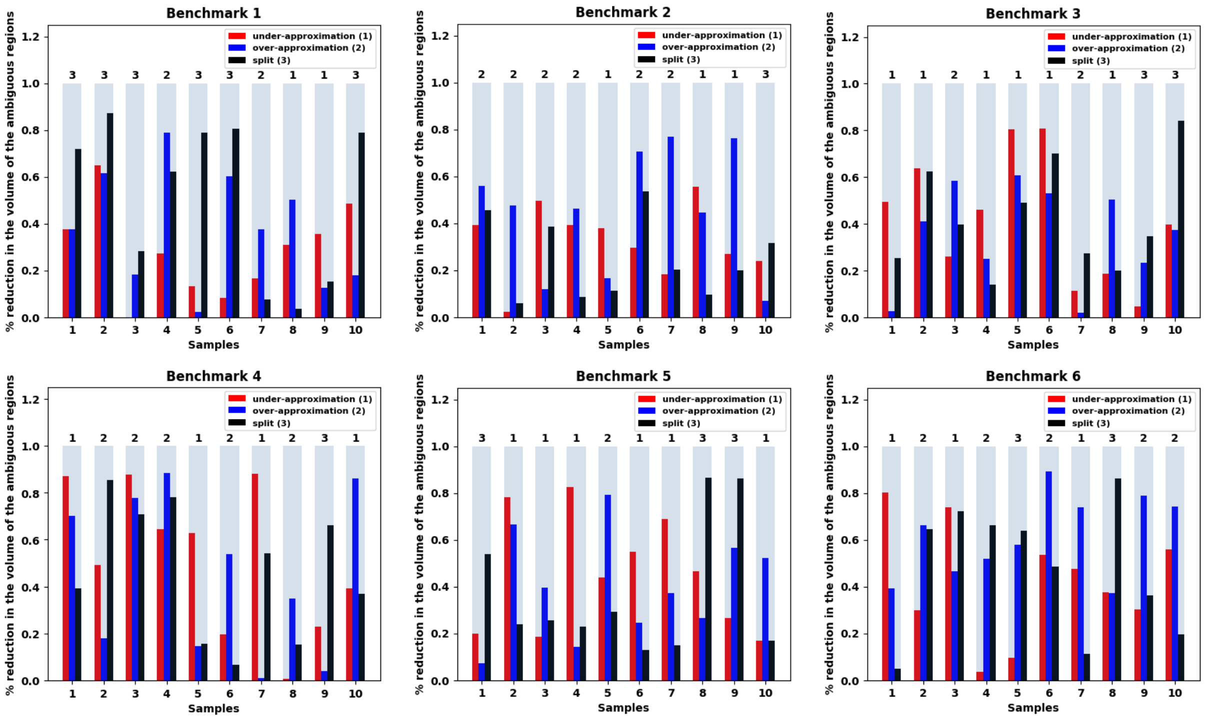


Figure 2.4: Percentage in reduction of the volume of ambiguous regions along with the NN output number (the number is at the top of histograms) for 20 samples for 6 evaluation benchmarks described in Table II.

$\frac{\text{data\_sample}-\mu}{\sigma}$, where μ and σ are the mean of the data and its standard deviation. The μ and σ parameters are initialized with respectively the empirical mean and standard deviation of the dataset and they are computed offline before the training.

2.8.2 NN's evaluation

We evaluated the NN on 6 different benchmarks as follows:

- In the first and second benchmarks, we generate the same random quadratic polynomials but in a different domain: $[-4, 4]^2$ and $[-10, 10]^2$. This choice is made to test the generalization of the NN outside the data domains that were used in its training. This is important since the Bernstein coefficients of a polynomial (the input to the NN) depend on the input region I_n.

Table 2.2: Evaluation of the trained NN on the six different benchmarks.

Benchmark	$p(x)$	n	order	region	Accuracy
1	$c_1x_1^2 + c_2x_2 + c_3x_1x_2 + c_4x_2 + c_5y + c_6$	2	2	$[-4, 4]^2$	95%
2	$c_1x_1^2 + c_2x_2 + c_3x_1x_2 + c_4x_2 + c_5y + c_6$	2	2	$[-10, 10]^2$	93%
3	$c_1x_1^4 + c_2x_2^3 + c_3x_1^4x_2^3 + c_4x_1^3 + c_5$	2	4	$[-2, 2]^2$	88%
4	$c_1x_1^{10} + c_2x_2^5 + c_3x_1^5x_2^3 + c_4x_1^5 + c_5$	2	10	$[-2, 2]^2$	87%
5	$c_1x_1^3 + c_2x_2^3 + c_3x_3^3 + c_4x_4^3$	4	3	$[-2, 2]^4$	81%
6	$c_1x_1^3 + c_2x_2^3 + c_3x_3^3 + c_4x_4^3 + c_5x_5^3 + c_6x_6^3 + c_7x_7^3$	7	3	$[-2, 2]^7$	80%

- In the third and fourth benchmarks, we generated random polynomials with degrees 4 and 10 over the domain $\big[-2, 2\big]^2$. These benchmarks are used to validate the generalization of the NN to polynomials of orders higher than the ones used in its training.

- Finally, in the fifth and sixth benchmarks, we generated random polynomials with higher dimensions, i.e., with dimension $n = 4$ and $n = 7$.

In summary, these benchmarks will help us to answer the following question: can the trained NN generalize to new data with different domains (benchmarks 1 and 2), higher orders (benchmarks 3 and 4), and higher dimensions (benchmarks 5 and 6)? More detail about the different benchmarks is shown in Table 2.2.

Figure 2.4 shows the performance of the trained NN over 20 random samples of each of the six benchmarks. For each sample, we used the framework in [106] to compute the ground-truth percentage in the reduction of the volume of ambiguous regions after applying every action under-approximation, over-approximation, or split. We then evaluated the NN on each sample and reported in Figure 2.4 both the ground-truth reduction of the ambiguous regions (as bars) against the index of the action suggested by the NN (as the text above the bars). As it can be seen from Figure 2.4, except for the second sample of the first benchmark, the NN outputs represent the actions that lead to the maximum reduction of the ambiguous region's volume.

Finally, we ran the same experiment for 1000 samples and report the percentage of samples for which the NN was able to predict the action that leads to the maximum reduction in the ambiguous region's volume. As it can be seen from Table 2.2, the trained NN is able to generalize on the different benchmarks. For instance, evaluating the NN on different domains results in the lowest accuracy of 93%. Furthermore, evaluating the NN on polynomials with higher-order results in an accuracy of 87%. Finally, the NN achieves 80% on higher dimension benchmarks.

2.9 Numerical Results - Scalability Results

In this section, we study the scalability of PolyARBerNN in terms of execution times by varying the order, the number of variables, and the number of the polynomial constraints for instances when a solution exists (the problem is Satisfiabile or SAT for short) and when a solution does not exist (or UNSAT for short). We will perform this study in comparison with state-of-the-art solvers including our previous solver PolyAR, Z3 8.9, and Yices 2.6. Next, we compare the performance of PolyARBerNN against a theorem prover named PVS which implements a Bernstein library to solve multivariate polynomial constraints [70].

Finally, we compare the scalability of the PolyAROpt optimizer against the built-in optimization library in Z3 8.9 to solve an unconstrained multivariate polynomial optimization problem with varying order and number of variables.

2.9.1 Scalability of PolyARBerNN against other SMT Solvers

In this experiment, we compare the execution times of PolyARBerNN against the PolyAR tool [106], Z3 8.9, and Yices 2.6. We consider two instances of Problem 1: an UNSAT and SAT problems. For each instance, we consider three scenarios, $m = 1$, $m = 5$, and $m = 10$

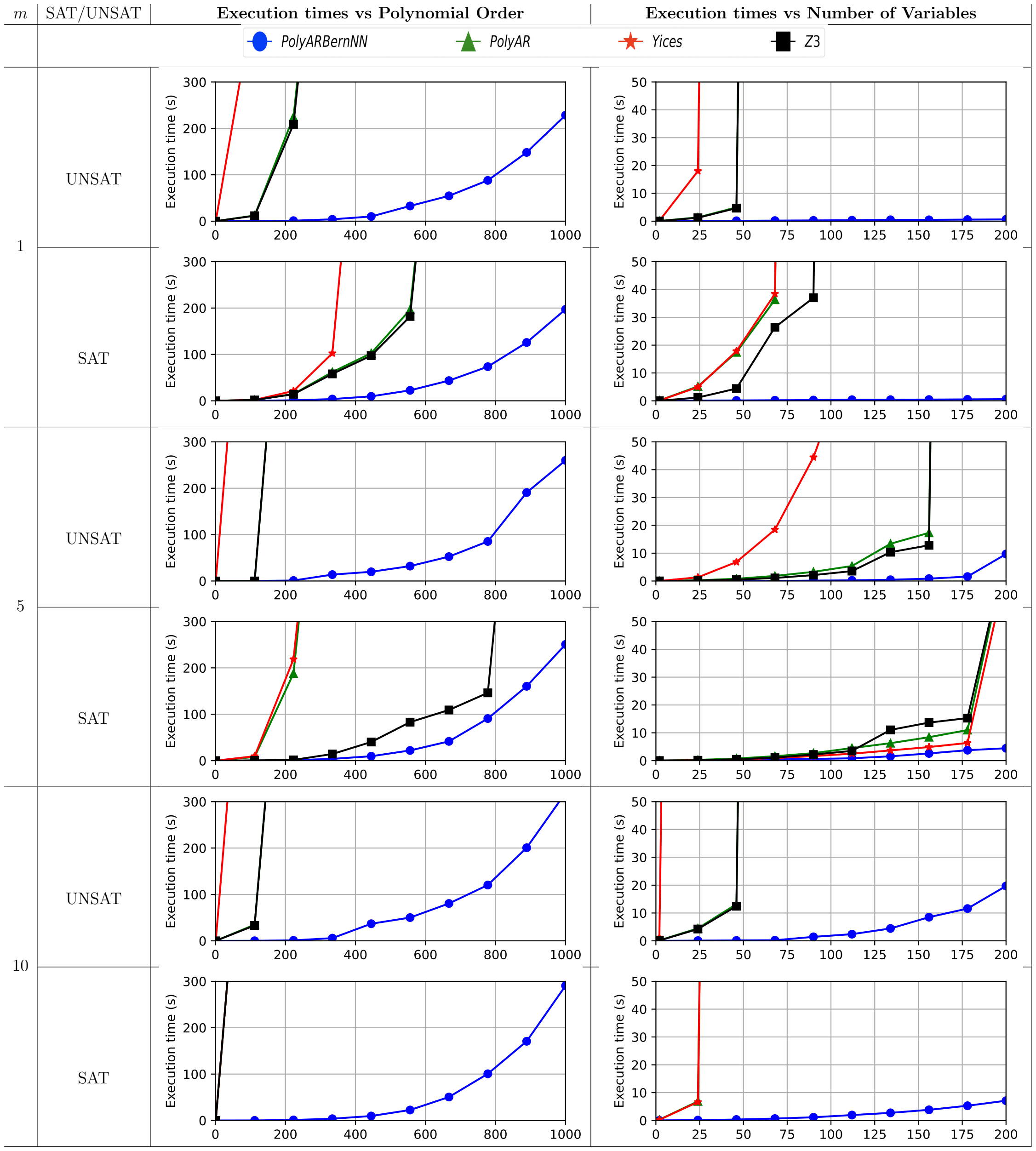


Figure 2.5: Scalability results of PolyARBerNN in the UNSAT case for 1, 5, and 10 constraints. (left) evolution of the execution time in seconds as a function of the order of the polynomials, (right) evolution of the execution time in seconds as a function of the number of variables. The timeout is equal to 1 hour..

where m is the number of polynomial constraints. First, we vary the order of the polynomials from 0 to 1000 while fixing the number of variables (and hence the dimension of the search space) to two. Alternatively, we also fix the order of the polynomials to 30 while varying the number of variables from 1 to 200.

We set the timeout of the simulations to be 1 hour. Figure 2.5 reports the execution times for all the experiments whenever the problem is UNSAT and SAT.

As evidenced by the figures, PolyARBerNN succeeded to solve the instances of Problem 1 for all orders and numbers of variables in a few seconds. For instance, solving 10 polynomial constraints with 200 variables and a maximum order of 30 took around 20 s leading to a speed-up of 200$\times$ compared to Z3 and Yices. On the other hand, other solvers are incapable of solving the polynomial constraints for all orders or number of variables and they time out after one hour.

These results show the scalability of the proposed approach by including Bernstein coefficients to prune the search space and a NN to guide the abstraction refinement.

2.9.2 Scalability of PolyARBerNN against other Bernstein-based solvers

PolyARBerNN was compared against Bernstein-based solvers such as PVS [70] and Realroot [69] on computing a root/solution for the following multivariate polynomial equation system. The scalability was evaluated by fixing the number of variables, changing the maximum order, fixing the maximum order, and varying the number of variables. PolyARBerNN successfully solved the systems for all orders and variables, while Realroot and PVS encountered timeouts. The execution times are shown in Figure 2.6 for both scenarios, with PolyARBerNN outperforming the other solvers.

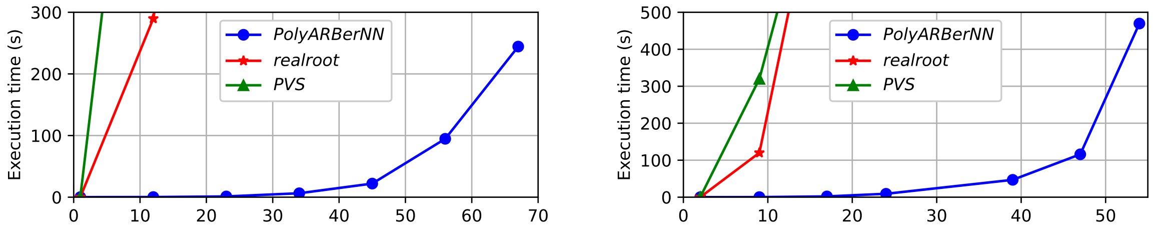


Figure 2.6: Scalability results of PolyARBerNN for multivariate polynomial equation system over the interval $I_n = [-1, 1]^n$. (left) evolution of the execution time in seconds as a function of the order of the polynomial with the number of variables $n = 3$. (right) evolution of the execution time in seconds as a function of the number of variables with maximum order equal to 3. The timeout is equal to 1 hour.

2.9.3 Scalability of PolyAROpt against other solvers

We compare the scalability results of PolyAROpt with the Z3 solver since Z3 has a built-in optimization library, Bernstein-based optimizer Borderbasix [93], and SOSTool [1]. Unfortunately, Yices does not have such an optimizer. We set the timeout of the experiment to be 1 hour. Figure 2.7 reports the execution times of two experiments that compute unconstrained optimization's minimum and maximum. As evidenced by the two figures, PolyAROpt succeeded in solving the unconstrained optimization problem for all orders and numbers of variables. For instance, solving the unconstrained optimization problem with 70 variables and a maximum order of 3 took around 50 *seconds*. On the other hand, the Z3 Borderbasix, SOSTool solvers cannot solve the unconstrained optimization problem for all orders or number of variables and time out.

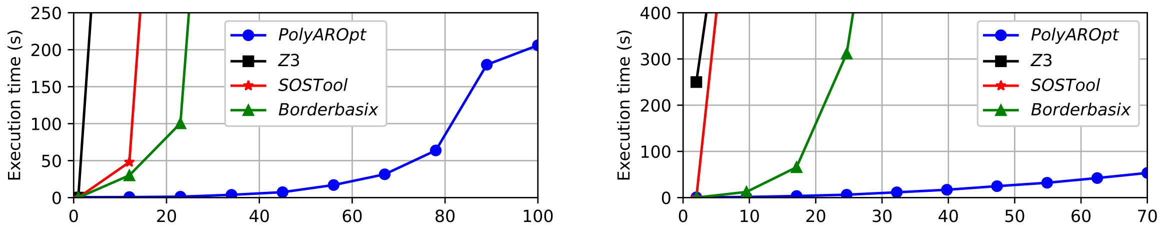


Figure 2.7: Scalability results of PolyAROpt for unconstrained optimization over the interval $I_n = [-1, 1]^n$. (left) evolution of the execution time in seconds as a function of the polynomial order. (right) evolution of the execution time in seconds as a function of the number of variables. The timeout is equal to 1 hour.

2.10 Numerical Results - Use Cases

In this section, we provide two engineering use cases. The first one focuses on the use of PolyARBerNN to synthesize stabilizing non-parametric controllers for nonlinear dynamical systems. The second use case focuses on the use of PolyAROPT to perform reachability analysis of polynomial dynamical systems.

2.10.1 Use Case 1: Nonlinear Controller Design for a Duffing Oscillator

In this subsection, we assess the scalability of the PolyARBerNN solver compared to state-of-the-art solvers for synthesizing a non-parametric controller for a Duffing oscillator reported by [42]. All the details of the dynamics of the oscillator and how we generated the polynomial constraints can be found in chapter 1. We denote by n the dimension of the Duffing oscillator.

We consider two instances of the controller synthesis problem for the Duffing oscillator with the following parameters:

- $n = 3$, $\zeta = 1.0$, $x(0) = [0.15, 0.15, 0.15]$, $L_1(x(k), u(k)) = (-x_1^3(k) + x_3^3(k) + u(k) - 2)^{51}$, $L_2(x(k), u(k)) = x_1^{51}(k)\, x_3^7(k) + x_1^9(k)\, x_3^5(k) - 5x_2^4(k) - x_2^2(k)\, u^2(k)$, which results in 9 polynomial constraints with 4 variables and max polynomial order of 153.

- $n = 4$, $\zeta = 1.75$, $x(0) = [0.1, 0.1, 0.01, 0.1]$, $L_1(x(k), u(k)) = x_1^4(k) + x_2^4(k) + x_3^4(k) + x_4^4(k) - u^4(k)$, $L_2(x(k), u(k)) = -x_1^{51}(k)\, x_3^{20}(k) - 5x_2^4(k) - x_2^2(k)\, u^2(k)$, $L_3(x(k), u(k)) = (x_1 x_2^2 - u(k) - 100)^{41}$, which results in 12 polynomial constraints with 5 variables and max polynomial order of 82.

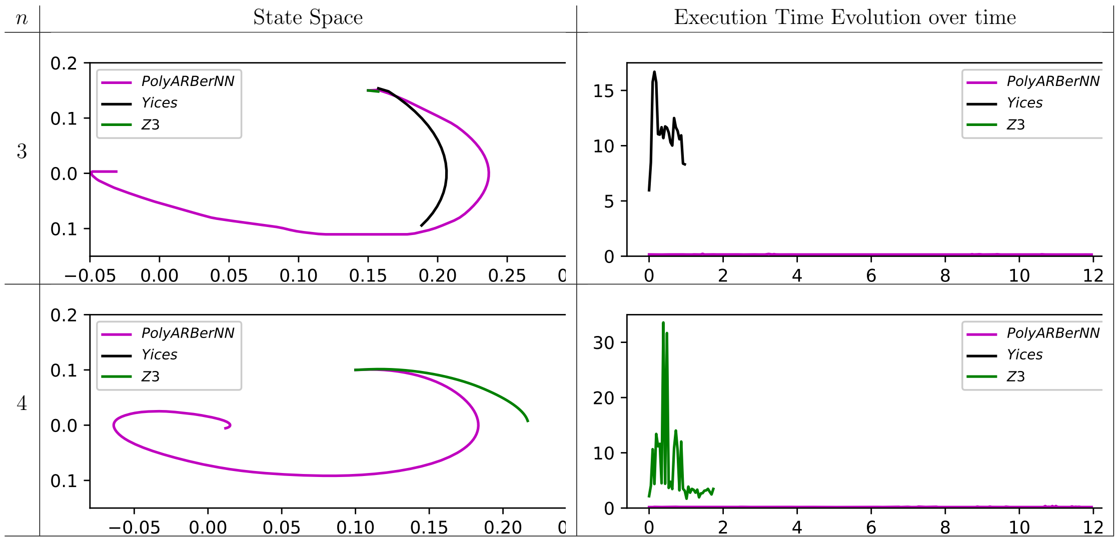


Figure 2.8: Results of controlling the Duffing oscillator with different n (left) evolution of the states $x_1(k)$ and $x_2(k)$ for the solvers in the state-space, (right) evolution of the execution time of solvers during the 12 seconds. The timeout is equal to $60s$. Trajectories are truncated once the solver exceeds the timeout limit.

We feed the resultant polynomial inequality constraint to PolyARBerNN, Yices, and Z3. We solve the feasibility problem for $n = 3$ and $n = 4$. We set the timeout to be $60s$. Figure 2.8 (left) shows the state-space evolution of the controlled Duffing oscillator for different solvers for number of variables n of 3 and 4. Figure 2.8 (right) shows the evolution of the execution time of the solvers during the 12 seconds. As it can be seen from Figure 7, our solver PolyARBerNN succeeded to find a control input u that regulates the state to the origin for all n. However, off-the-shelf solvers are incapable of solving all two instances, and they early time out after 60 *seconds* out of the simulated 12 *seconds*. This shows the scalability of the proposed approach.

2.10.2 Use Case 2: Reachability analysis of a discrete polynomial dynamical systems

In this section, we show how to use PolyAROpt to compute the reachable set of states for discrete-time polynomial systems. We consider a discrete polynomial dynamical system of the following form:

$$x_{k+1} = f\left(x_k\right), \ k \in \mathbb{N}, x_0 \in Q_0, \tag{2.28}$$

where $f : \mathbb{R}^n \to \mathbb{R}^n$ is a multivariate polynomial map of a maximum degree $L = (l_1, \cdots, l_n)$, and Q_0 is a bounded polyhedron in $\mathbb{R}^n$. In this subsection, we consider a bounded-time reachability analysis of the system in (2.28). Computing the exact reachability sets of this type of dynamic system is hard. Therefore, we overapproximate the exact set with a simplified set such as bounded polyhedra. Bounded polyhedra are easy to handle and analyze. Computing the reachability set, after a finite time K, involves computing sequentially the reachability set at every time step k using the following relation $Q_{k+1} = f\left(Q_k\right), k = 0, \cdots, K-1$, where Q_k is a bounded polyhedra. A bounded polyhedra is represented with an H-representation $Q_k = (A_k, b_k) = \{x \in \mathbb{R}^n | A_k x \leq b_k\}$, where the inequality is a point-wise inequality. The template A_k represents the directions of Q_k's faces and b_k represents their positions. Given a polyhedra $Q_k = (A_k, b_k)$, we need to compute $Q_{k+1} = f\left(Q_k\right)$. We assume that $A_k \in \mathbb{R}^{m \times n}$ is given. Now, we need to compute $b_{k+1} \in \mathbb{R}^m$ which can be obtained through the following optimization problem [12]:

$$-b_{k+1,i} \leq u_{k+1,i} = \min_{x \in Q_k} -A_{k+1,i} f\left(x\right), \forall i = 1, \cdots, m, \tag{2.29}$$

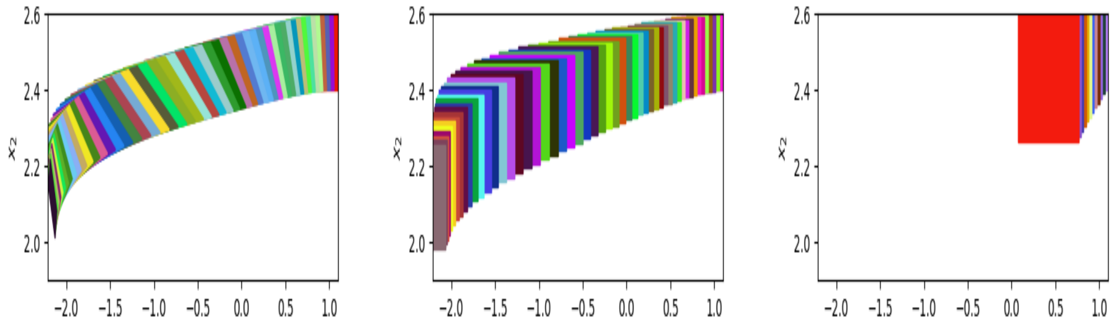


Figure 2.9: Reachability computation for the FitzHugh-Nagumo neuron model. **Left:** using PolyAROpt for number of steps $K = 50$. **Center:** using Sapo for number of steps $K = 50$. **Right:** using Flowstar for number of steps $K = 50$.

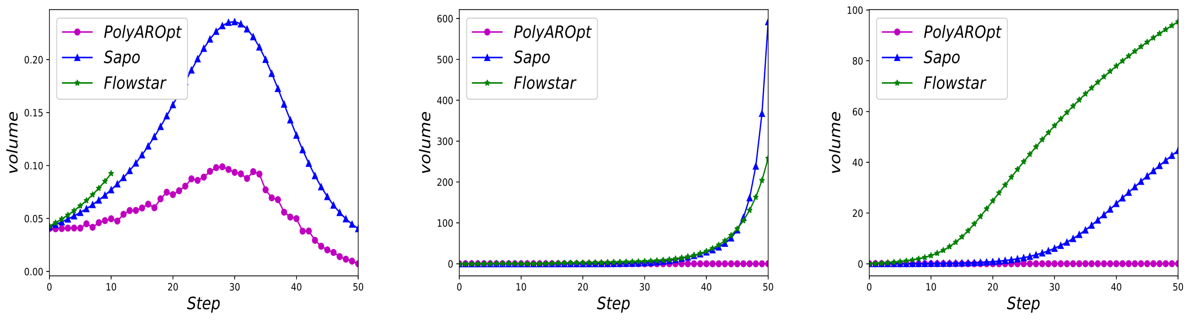


Figure 2.10: The volume of the reachable set of states that are obtained using PolyAROpt, Sapo, and Flowstar (left) FitzHugh-Nagumo, (center) Duffing oscillator, and (right) Jet flight.

where $A_{k+1,i}$ and $b_{k+1,i}$ are the i^{th} row and component of the templates A_{k+1} and b_{k+1}, respectively. In every step $k \in \mathbb{N}$, we compute an upper bound for $-b_{k+1,i}$, by solving the optimization problem (2.29) using PolyAROpt and then an overapproximation of the reachability set Q_{k+1} is computed. In order to use PolyAROpt to solve (2.29), we need to overapproximate the polyhedra Q_k with a hyperreactangle R_k. Therefore, the optimization problem (2.29) is modified as follows:

$$-b_{k+1,i} \leq u_{k+1,i} = \min_{x \in R_k} -A_{k+1,i} f(x), \forall i = 1, \cdots, m,$$
$$\text{subject to } x \in Q_k. \tag{2.30}$$

We implemented the reachability computation method and tested it on three dynamical systems:

- FitzHugh-Nagumo Neuron: is a polynomial dynamic systems that models the electrical activity of a neuron [41]. We performed reachability analysis for $K = 50$ time steps with the initial set of states $Q_0 = [0.9, 1.1] \times [2.4, 2.6]$,
- Duffing Oscillator: is a discrete-time version of a nonlinear oscillator model [106]. We performed reachability analysis for $K = 50$ time steps with the initial set of states

$Q_0 = [2.49, 2.51] \times [1.49, 1.51]$,

- Jet Flight: is a discrete-time version of a jet flight model [18]. We performed reachability analysis for $K = 50$ time steps with the initial set of states $Q_0 = [0.9, 1.2] \times [0.9, 1.2]$.

All the details of the dynamics of the three dynamical systems and how we generated the polynomial constraints can be found in [12, 41, 106, 18]. We computed the reachable sets for these dynamical systems and compared our results with Sapo [25] and Flowstar [19]. Sapo is a tool proposed for the reachability analysis of the class of discrete-time polynomial dynamical systems. Sapo linearizes the optimization problem (2.29) using the Bernstein form of the polynomial and was shown to outperform state-of-the-art reachability analysis tools like Flowstar [19]. Flowstar [19] is a tool used in the ARCH workshop competition for hybrid systems' reachability. This tool is a state-of-the-art reachability analysis that uses Taylor models to compute the reachable sets. Figure 2.9 shows the reachable sets computed by PolyAROpt compared to Sapo and Flowstar for the FitzHugh-Nagumo Neuron model. Inspecting the results in Figure 2.9 qualitatively shows that PolyAROpt is capable of computing tighter sets compared to Sapo and Flowstar. Flowstar stopped the computation of reachable sets at the $10 - th$ step due to large overestimation errors. To quantitatively compare the results of PolyAROpt, Sapo, and Flowstar, we compute the volume of each reachable set (for different time steps). Figure 2.10 shows these volumes for all the three dynamical systems mentioned above. As evident by the results in Figure 2.10, PolyAROpt results in reachable sets that are tighter than the one obtained from the Sapo and Flowstar thanks to PolyAROpt's ability to solve the polynomial optimization problem without any relaxations or using Taylor models. Such ability to avoid relaxation results in several orders of magnitude reduction in the volume of the reachable sets compared to Sapo and Flowstar; a significant improvement in the analysis of such dynamical systems.

Part II

Designing scalable Model Checkers to analyze correctness of Deep Neural Networks

Chapter 3

BERN-NN: Tight Bound Propagation For Neural Networks Using Bernstein Polynomial Interval Arithmetic

In this chapter, we present BERN-NN as an efficient tool to perform bound propagation of Neural Networks (NNs). Bound propagation is a critical step in wide range of NN model checkers and reachability analysis tools. Given a bounded input set, bound propagation algorithms aim to compute tight bounds on the output of the NN. So far, linear and convex optimizations have been used to perform bound propagation. Since neural networks are highly non-convex, state-of-the-art bound propagation techniques suffer from introducing large errors. To circumvent such drawback, BERN-NN approximates the bounds of each neuron using a class of polynomials called Bernstein polynomials. Bernstein polynomials enjoy several interesting properties that allow BERN-NN to obtain tighter bounds compared to those relying on linear and convex approximations. BERN-NN is efficiently parallelized on graphic processing units (GPUs). Extensive numerical results show that bounds obtained by BERN-NN are orders of magnitude tighter than those obtained by state-of-the-art verifiers such as linear programming and linear interval arithmetic. Moreoveer, BERN-NN is both faster and produces tighter outputs compared to convex programming approaches like alpha-CROWN.

3.1 Introduction

Neural Networks (NNs) have become an increasingly central component of modern, safety-critical, cyber-physical systems like autonomous driving, autonomous decision-making in smart cities, and even autonomous landing in avionic applications. Thus, there is an increasing need to verify the safety and correctness [89, 90, 43] of NNs when they are used to control physical systems.

The problem of NN Verification has been well studied in literature [63]. Most NN verifiers rely mainly on either using linear relaxation and optimization [96, 30, 100, 51, 59, 98] to falsify a given property or prove its satisfaction, or reachability analysis to compute an over-approximation of the output set. The latter is specifically important for control applications where the property of interest is defined over a time horizon. Both techniques rely on overapproximation, hence, having tight output bounds is at the core of NN verification as it allows reasoning about NN properties in an efficient manner. For example, model checking the robustness of NNs against adversarial perturbations can be done by simply comparing the tight bounds of the outputs of the network. Moreover, networks used in control applications often involve multi-step reachability, and hence computing tight bounds is crucial to harness the accumulation of the error and hence be able to efficiently reason about the safety of the system.

Due to the non-convexity and non-linearity of NNs, the problem of finding the exact bounds of NN outputs is NP-hard[57]. Different tools have been proposed to find tight overapproximations of NN outputs. MILP-based methods [27, 14, 91, 8, 17, 40, 3, 20] encode the non-linear activations as linear and integer constraints. Reachability methods [102, 101, 49, 97, 92, 55, 37] use layer-by-layer reachability analysis (exact or overapproximation) of the network. Most of these methods either rely on convex *linear relaxation* of the non-linear activation functions to overapproximate the output of the NN, or try to find the exact bounds which are often intractable.

In this work, we explore using polynomials to approximate non-linear activations (e.g. ReLU). More specifically, we approximate non-linear activations using Bernstein polynomials which are constructed as a linear combination of the Bernstein basis polynomials [33]. The use of Bernstein polynomials is motivated by two reasons. First, based on the Stone-Weierstrass approximation theorem [22], Bernstein polynomials can uniformly approximate continuous activation functions. Second and most importantly, bounding a Bernstein polynomial is computationally cheap based on the interesting properties of Bernstein polynomials discussed in section 3.3. The goal of using higher-order polynomials versus linear relaxation is to get tight bounds on NNs which is crucial for verifying a large class of formal properties. This idea of using polynomials has inspired other researchers [29, 32, 53], however, the proposed tools suffer from scalability issues.

The main contributions of this chapter can be summarized as follows:

1. We propose a tool that uses Bernstein polynomials to approximate ReLU activations and hence compute tighter NN bounds than state-of-the-art.

2. The tool is designed with scalability in mind; hence, the entire operations can be accelerated using GPUs.

3. We show that by using the proposed approximation, we are able to compute tighter output sets than alpha-Crown (winner of VNN22' competition[5] for Formal Verification of NNs) and other state-of-the-art bounding methods. For instance, BERN-NN ap-

proximations are twice reduced compared to alpha-Crown for actual NN's controllers. Moreover, Numerical results showed that Bern-NN can process neural networks with more than 1000 neurons in less than 2 minutes

3.2 Problem Formulation

3.2.1 Notation:

General notation: We use the symbols $\mathbb{N}$ and $\mathbb{R}$ to denote the set of natural and real numbers, respectively. We denote by $x = \big(x_1, x_2, \cdots, x_n\big) \in \mathbb{R}^n$ the vector of n real-valued variables, where $x_i \in \mathbb{R}$. We denote by $I_n(\underline{d}, \overline{d}) = \big[\underline{d}_1, \overline{d}_1\big] \times \cdots \times \big[\underline{d}_n, \overline{d}_n\big] \subset \mathbb{R}^n$ the n-dimensional hyperrectangle where $\underline{d} = (\underline{d}_1, \cdots, \underline{d}_n)$ and $\overline{d} = \big(\overline{d}_1, \cdots, \overline{d}_n\big)$ are the lower and upper bounds of the hyperrectangle, respectively. We denote by x^T and A^T the transpose operation of the vector x and the matrix A. We denote by 0_n a vector that contains n zero values and by $0_{n\times m}$ the matrix of shape $n \times m$ that contains zeros. Finally, $A * B$ stands for the element-wise product between the multi-dimensional tensors A and B, and $A \otimes B$ stands for the Kronecker product between the matrices A and B.

Notation pertaining to multivariate polynomials: For a real-valued vector $x = \big(x_1, x_2, \cdots, x_n\big) \in \mathbb{R}^n$ and an index-vector $K = (k_1, \cdots, k_n) \in \mathbb{N}^n$, we denote by $x^K \in \mathbb{R}$ the scalar $x^K = x_1^{k_1} \times \ldots \times x_n^{k_n}$. Given two multi-indices $K = (k_1, \cdots, k_n) \in \mathbb{N}^n$ and $L = (l_1, \cdots, l_n) \in \mathbb{N}^n$, we use the following notation throughout this paper:

$$K + L = (k_1 + l_1, \cdots, k_n + l_n),$$
$$\binom{L}{K} = \binom{l_1}{k_1} \times \cdots \times \binom{l_n}{k_n},$$
$$\sum_{K \le L} = \sum_{k_1 \le l_1} \cdots \sum_{k_n \le l_n}$$

Finally, a real-valued multivariate polynomial $p : \mathbb{R}^n \to \mathbb{R}$ is defined as:

$$\begin{aligned} p(x_1, \ldots, x_n) &= \sum_{k_1=0}^{l_1} \sum_{k_2=0}^{l_2} \cdots \sum_{k_n=0}^{l_n} a_{(k_1,\ldots,k_n)} x_1^{k_1} x_2^{k_2} \ldots x_n^{k_n} \\ &= \sum_{K \le L} a_K x^K, \end{aligned}$$

where $L = (l_1, l_2, \ldots, l_n)$ is the maximum degree of x_i for all $i = 1, \ldots, n$.

Notation pertaining to neural networks: In this paper, we consider H-layer, feed-forward, ReLU-based neural networks $\mathcal{NN} : \mathbb{R}^n \to \mathbb{R}^o$ defined as:

$$\begin{aligned} \mathcal{NN}(x) &= W^{(H)} z^{(H-1)} + b^{(H)} \\ z^{(H-1)} &= \sigma\left(W^{(H-1)} z^{(H-2)} + b^{(H-1)}\right) \\ &\vdots \\ z^{(1)} &= \sigma\left(W^{(1)} x + b^{(1)}\right) \end{aligned}$$

where σ is the ReLU activation function (i.e., $\sigma(z) = \max(0, z)$) that operates element-wise, $W^{(i)} \in \mathbb{R}^{h_i \times h_{i-1}}$ and $b^{(i)} \in \mathbb{R}^{h_i}$ with $i \in \{1, \cdots, H\}$ are the weights and the biases of the network. For simplicity of notation, we use $\hat{z}_j^{(i)}$ and $z_j^{(i)}$ to denote the pre-activation (input) and the post-activation (output) of the j-th neuron in the i-th layer.

3.2.2 Main Problem:

In this paper, we seek to find polynomials that upper and lower approximate the NN's outputs $\mathcal{NN}(x)$ whenever the NN's input x is confined within a pre-defined hypercube, i.e. $x \in I_n(\underline{d}, \overline{d})$.

Problem 3.1. *Given a neural network $\mathcal{NN} : \mathbb{R}^n \to \mathbb{R}^o$ and an input domain hypercube $I_n(\underline{d}, \overline{d}) \subset \mathbb{R}^n$. Find lower and upper approximate polynomials $\left(\underline{p}_{\mathcal{NN},1}(x), \overline{p}_{\mathcal{NN},1}(x)\right)$*

$, \dots \left(\underline{p}_{\mathcal{NN},o}(x), \overline{p}_{\mathcal{NN},o}(x)\right)$*, such that:*

$$\underline{p}_{\mathcal{NN},1}(x) \leq \mathcal{NN}_1(x) \leq \overline{p}_{\mathcal{NN},1}(x)$$
$$\vdots$$
$$\underline{p}_{\mathcal{NN},o}(x) \leq \mathcal{NN}_o(x) \leq \overline{p}_{\mathcal{NN},o}(x),$$

where with some abuse of notation, we use $\mathcal{NN}_i(x)$ to denote the ith output of the neural network $\mathcal{NN}$.

Note that the lower/upper bound polynomials $\left(\underline{p}_{\mathcal{NN},1}(x), \overline{p}_{\mathcal{NN},1}(x)\right), \dots \left(\underline{p}_{\mathcal{NN},o}(x), \overline{p}_{\mathcal{NN},o}(x)\right)$ depend on the input domain I_n. That is, for each value of I_n, we need to find different lower/upper bound polynomials. However, for the sake of simplicity of notation, we drop the dependency on I_n.

3.3 Tight bounds of ReLU Functions Using Bernstein Polynomials

To solve Problem 3.1, we rely on a class of polynomials called Bernstein polynomials which are defined as follows:

Definition 3.1. *(Bernstein Polynomials) Given a continuous function $g : \mathbb{R}^n \to \mathbb{R}$, an input* *domain (hypercube) $I_n(\underline{d}, \overline{d}) \subset \mathbb{R}^n$, and a multi-index $L = (l_1, \cdots, l_n) \in \mathbb{N}^n$, the polynomial:*

$$B_{g,L}(x) = \sum_{K \le L} b^g_{K,L} Ber_{K,L}(x), \tag{3.1}$$

$$Ber_{K,L}(x) = \binom{L}{K} \frac{(x - \underline{d})^K \left(\overline{d} - x\right)^{L-K}}{\left(\overline{d} - \underline{d}\right)^L}, \tag{3.2}$$

$$b^g_{K,L} = g\left(\left(\overline{d}_1 - \underline{d}_1\right) \frac{k_1}{l_1} + \underline{d}_1, \cdots, \left(\overline{d}_n - \underline{d}_n\right) \frac{k_n}{l_n} + \underline{d}_n \right), \tag{3.3}$$

is called the Lth order Bernstein polynomial of g, where $Ber_{K,L}(x)$ and $b^g_{K,L}$ are called the Bernstein basis and Bernstein coefficients of g, respectively.

Bernstein polynomials are known to be capable of approximating any continuous function. That is, Bernstein approximation has an advantage compared to Taylor approximation because the latter relies on the function being differentiable. In this case, Taylor model can not approximate ReLU activation functions because they are not differentiable which makes Bernstein polynomials a good option to approximate ReLU functions. Bernstein polynomials have an interesting and useful property called *range enclosing property* which is defined as follows:

Definition 3.2. *(Range Enclosing Property [86]) Given a multi-dimensional polynomial $p(x)$ of order L that it defined over the region $I_n\left(\underline{d}, \overline{d}\right)$ with its Bernstein polynomial $B_{p,L} = \sum_{K \le L} b^p_{K,L}(x) Ber_{K,L}(x)$. The following holds for all $x \in I_n\left(\underline{d}, \overline{d}\right)$:*

$$\min_{K \le L} b^p_{K,L} \le p(x) \le \max_{K \le L} b^p_{K,L}. \tag{3.4}$$

The range enclosing property states that the minimum (maximum) over all the Bernstein coefficients is a lower (upper) bound for the polynomial p over the region $I_n\left(\underline{d}, \overline{d}\right)$. These bounds provided by the Bernstein coefficients are generally tighter than those given by interval arithmetic and many centered forms [87]. Note that the range enclosing property

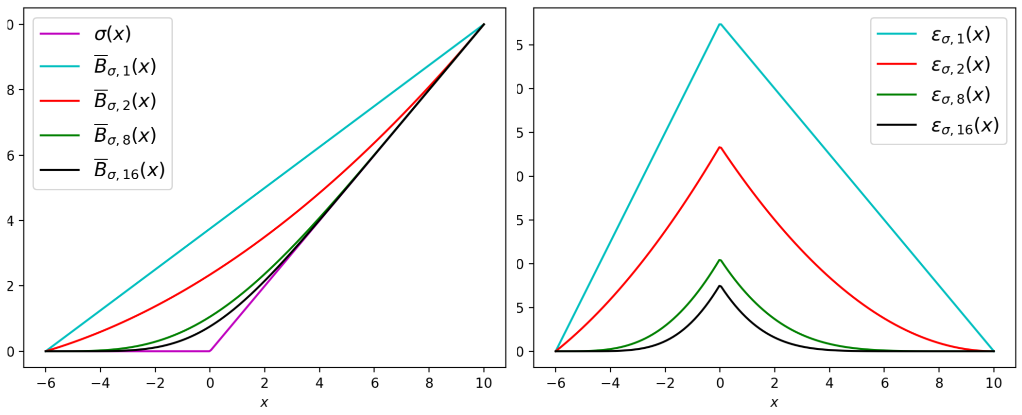


Figure 3.1: **(Left)** Bernstein polynomial approximations of ReLU activation for different approximation's order $L \in \{1, 2, 8, 16\}$, in the interval $I_1(-6, 10) = \big[-6, 10\big]$. **(Right)** Bernstein polynomial approximations of ReLU and their associated approximation errors for different approximation's order $L \in \{1, 2, 8, 16\}$ in the interval $I_1(-6, 10) = \big[-6, 10\big]$.

applies only when the Bernstein polynomial is used to approximate other polynomials p and other continuous functions g. Nevertheless, as we show in Section 4, these bounds will be helpful to provide tight bounds on the polynomials used to over/under approximate the individual neurons and hence obtain tight polynomial bounds on the NN's outputs.

3.3.1 Over-Approximating ReLU functions using Bernstein Polynomials

We now study how to use Bernstein polynomials to over-approximate the ReLU function $\sigma : \mathbb{R} \to \mathbb{R}$ defined as $\sigma(x) = \max(0, x)$. While Bernstein polynomials can approximate any continuous function g, there is no guarantee that this Bernstein approximation is either over-approximation or under-approximation. The next result establishes an order between the ReLU function σ and its Bernstein approximation.

Proposition 3.1. *Given an interval* $I_1\left(\underline{d}, \overline{d}\right) = \left[\underline{d}, \overline{d}\right]$, *where* $0 \in \left[\underline{d}, \overline{d}\right]$ *and any approxima-*

tion order $L \geq 1$. The following holds for all $x \in I_1$:

$$\sigma(x) \leq B_{\sigma,L}(x) = \overline{B}_{\sigma,L}(x).$$

Proof. This follows directly by substituting the function σ in the definition of Bernstein polynomials (4.1)-(3.3). □

In other words, Proposition 3.1 states that the Bernstein polynomial of σ is a guaranteed over-approximation of σ. This even holds *for any approximation order L*. Moreover, since the approximation error between a function g and its Bernstein approximation $B_{g,L}$ is known to decrease as L increases [44]. Then another consequence of Proposition 3.1 is that Bernstein polynomials produce a tighter over-approximation for ReLU functions as L increases.

Figure 3.1 emphasizes these conclusions pictorially where we show the Bernstein polynomials of σ with orders $L = 1, 2, 8, 16$. As shown in Figure 3.1 (Left), the Bernstein polynomials $B_{\sigma,L}(x)$ for $L \in \{1, 2, 8, 16\}$ over-approximate the ReLU activation function over the entire input range. Furthermore, the over-approximation gets tighter to the actual ReLU by increasing the approximation order L. We note that using $L = 1$, the resulting Bernstein polynomial produces the well-studied linear convexification of the ReLU function which is used in state-of-the-art algorithms for bounding neural networks including Symbolic Interval Arithmetic (SIA) [96] and alpha-CROWN [103]. In other words, Bernstein polynomials can be seen as a generalization of these techniques.

3.3.2 Under-approximating ReLU functions using Bernstein polynomials

In addition to the over-approximation of the ReLU function σ, it is essential to establish a Bernstein under-approximation of σ which is captured by the following result.

Proposition 3.2. *Given an interval* $I_1\left(\underline{d},\overline{d}\right)=\left[\underline{d},\overline{d}\right]$, *where* $0\in\left[\underline{d},\overline{d}\right]$, *then the following holds for all* $x\in I_1$*:*

$$\underline{B}_{\sigma,L}(x)=\overline{B}_{\sigma,L}(x)-\overline{B}_{\sigma,L}(0)\leq\sigma(x).$$

Proof. To prove the result, we define the approximation error $\epsilon_{\sigma,L}$ as:

$$\epsilon_{\sigma,L}(x)=\overline{B}_{\sigma,L}(x)-\sigma(x).$$

We bound the maximum estimation error satisfies as follows:

$$\begin{aligned}
\max_{x\in[\underline{d},\overline{d}]}\epsilon_{\sigma,L}(x)&=\max_{x\in[\underline{d},\overline{d}]}\left(\overline{B}_{\sigma,L}(x)-\sigma(x)\right) && (3.5)\\
&\overset{(a)}{=}\max_{x\in[\underline{d},0]}\overline{B}_{\sigma,L}(x) && (3.6)\\
&\overset{(b)}{=}\overline{B}_{\sigma,L}(0) && (3.7)
\end{aligned}$$

where (a) follows from the fact that $\sigma(x)=0$ for $x\in[\underline{d},0]$ and $\sigma(x)\geq 0$ for $x\in[0,\overline{d}]$ and hence the maximum of the equation is attained whenever $\sigma(x)=0$. Equation (b) holds from the monotonicity of $\overline{B}_{\sigma,L}(x)$ when $x\in[\underline{d},0]$—the monotonicity follows directly from the definition of $\overline{B}_{\sigma,L}(x)$—and hence the maximum is attained when $x=0$. It follows from the definition of $\epsilon_{\sigma,L}(x)$ that:

$$\begin{aligned}
\sigma(x)=\overline{B}_{\sigma,L}(x)-\epsilon_{\sigma,L}(x)&\geq\overline{B}_{\sigma,L}(x)-\max_{x\in[\underline{d},\overline{d}]}\epsilon_{\sigma,L}(x)\\
&=\overline{B}_{\sigma,L}(x)-\overline{B}_{\sigma,L}(0)=\underline{B}_{\sigma,L}
\end{aligned}$$

which concludes the proof. □

Proposition 3.2 shows that the maximum error between the Bernstein over-approximation polynomial $\overline{B}_{\sigma,L}$ and the ReLU activation function σ is equal to the value of the Bernstein polynomial at 0, i.e., $\overline{B}_{\sigma,L}(0)$. This result has a direct consequence on the efficiency of our tool. It is enough to propagate over-approximation of the ReLU function and one can get an under-approximation directly by shifting the over-approximation polynomial.

Figure 3.1 (Right) emphasizes this fact pictorially. As it is shown in the figure, the maximum error $\epsilon_{\sigma,L}(x) = \overline{B}_{\sigma,L} - \sigma(x)$ is reached at $x = 0$ and is equal to $\overline{B}_{\sigma,L}\,(0)$.

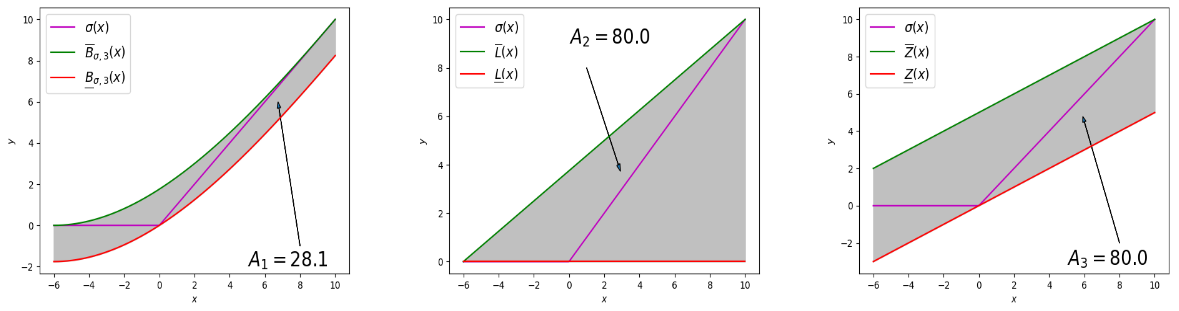


Figure 3.2: Illustrations of the over-approximation sets (shaded in gray) of the ReLU activation functions in the interval $\big[-6, 10\big]$ using different approaches: Bernstein approach (Left), triangulation approach (Center), and zonotope approach (Right). Green (Red)-colored curves represent the over-approximation (under-approximation) curves for every approach, respectively. A_i, $i \in \{1, 2, 3\}$, represents the over-approximation set's area for every approach.

Table 3.1: The area of the over-approximation set of the ReLU activation functions in the interval $\big[-6, 10\big]$ using different Bernstein approach for different approximation order L.

Approx. Method	Triangulation	Zonotope	Bernstein poly		
			$L = 2$	$L = 3$	$L = 8$
error	80.0	80.0	37.5	28.1	16.9

3.3.3 Comparing Bernstein Approximation Against Widely Used Approximations

The major advantage of using Bernstein polynomials is that they produce a tighter approximation for the response function of ReLU compared to the other state-of-the-art techniques. In particular, existing techniques focus on "convexifying" the response of the ReLU function through linear approximation/triangulation (Figure 4.1-middle) or zonotopes (Figure

4.1-right). Unlike these techniques, Bernstein polynomials lead to tighter non-convex approximations of the non-convex ReLU function. While it is direct to obtain a closed-form expression for the difference in the approximation error between Bernstein polynomials and triangulation/zonotope approximations, we, instead support our conclusions with the numerical example shown in Table 3.1 and highlighted in Figure 4.1. In this example, we compute the approximation error (highlighted in gray) which captures the quality of the over and under-approximations. As captured by this example, it is direct to see that Bernstein polynomials lead to tighter approximation. Moreover, such approximation gets tighter as the approximation order L increases.

3.4 Encoding Basic Bernstein Polynomial Operations Using Multi-Dimensional Tensors

While using Bernstein polynomials to approximate individual ReLU functions provides tighter bounds compared to other techniques, computing Bernstein polynomials via its definition in (4.1)-(3.3) is time-consuming. That is why state-of-the-art techniques have focused on linear (or convex) relaxations to obtain tractable computations. Nevertheless, in this section, we show that technological advances in Graphics Processing Units (GPUs) can be used to perform all the required operations to efficiently compute Bernstein polynomial approximations of individual neurons along with propagating these polynomials from one layer of the neural network to the next layer. Our main contribution of this section is to encode all necessary operations over Bernstein polynomials into additions and multiplication of multi-dimensional tensors that can be easily performed using GPUs.

3.4.1 Multi-dimensional tensor representation of Bernstein polynomials

We represent the Bernstein polynomial:

$$B_{g,L}(x) = \sum_{K \leq L} b^g_{K,L} Ber_{K,L}(x)$$

of function g and order L as a multi-dimensional tensor Ten$(B_{g,L})$ of n dimensions, and of a shape of $L = (l_1 + 1, \cdots, l_n + 1)$, where the $K = (k_1, \cdots, k_n)$ component of Ten$(B_{g,L})$ is equal to the Bernstein coefficient $b^g_{K,L}$. The multi-dimensional tensor Ten$(B_{g,L})$ represent all the Bernstein coefficients $b^g_{K,L}$ of g, $\forall K \leq L$.

Example 3.1. *Consider the two-dimensional Bernstein polynomial:*

$$B_{g,L}(x_1, x_2) = \sum_{k_1=0}^{2} \sum_{k_2=0}^{3} b^g_{(k_1,k_2),L} Ber_{(k_1,k_2),L}(x_1, x_2)$$

with orders $L = (2, 3)$. Its two-dimensional tensor representation is written as follows:

$$Ten(B_{g,L}) = \begin{bmatrix} b^g_{(0,0),L} & b^g_{(0,1),L} & b^g_{(0,2),L} & b^g_{(0,3),L} \\ b^g_{(1,0),L} & b^g_{(1,1),L} & b^g_{(1,2),L} & b^g_{(1,3),L} \\ b^g_{(2,0),L} & b^g_{(2,1),L} & b^g_{(2,2),L} & b^g_{(2,3),L} \end{bmatrix}. \tag{3.8}$$

In a similar manner, we represent a multi-dimensional polynomial of order L written in the power series form $p(x) = \sum_{K \leq L} a_K x^K$ as a multi-dimensional tensor Ten(p) of n dimensions, and of a shape of $L = (l_1 + 1, \cdots, l_n + 1)$, where the $K = (k_1, \cdots, k_n)$ component of Ten(p) is equal to the coefficient a_K.

3.4.2 Multiplication of two multi-variate Bernstein polynomials

Multiplying two polynomials represented in the power series form on GPUs has been widely studied in the literature. Unlike power series, multiplying two Bernstein polynomials need extra handling [81]. In this subsection, we propose how to encode the multiplication of Bernstein polynomials using GPU implementations that were designed for power-series polynomials.

Given two multivariate polynomials written in a power series form, $p_1 = \sum\limits_{K \leq L_1} a_K^1 x^K$ and $p_2 = \sum\limits_{K \leq L_2} a_K^2 x^K$, and their tensor representation, $\text{Ten}\left(p_1\right)$ and $\text{Ten}\left(p_2\right)$, we use an efficient algorithm [74] that performs multivariate polynomial multiplications. We denote by $\textbf{Prod}\left(\text{Ten}\left(p_1\right), \text{Ten}\left(p_2\right)\right)$ the tensor resulting from such multiplication, i.e.:

$$\text{Ten}\left(p_1 p_2\right) = \textbf{Prod}\left(\text{Ten}\left(p_1\right), \text{Ten}\left(p_2\right)\right).$$

Applying power-series-based algorithms to multiply two Bernstein polynomials produce incorrect results. Different algorithms were proposed for the case when the Bernstein polynomials are functions of one variable x_1 [35] and two variables x_1, x_2 [81]. Below, we generalize the procedure in [81] to account for Bernstein polynomials in n variables.

Proposition 3.3. *Given two multivariate Bernstein polynomials $B_{g_1,L_1}(x) = \sum\limits_{K \leq L_1} b_{K,L_1}^{g_1}$ $Ber_{K,L_1}(x)$ and $B_{g_2,L_2}(x) = \sum\limits_{K \leq L_2} b_{K,L_2}^{g_2} Ber_{K,L}(x)$. The tensor representation of the Bernstein polynomial $B_{g_1,L_1}(x) B_{g_2,L_2}(x)$ can be computed as follows:*

$$Ten\left(\tilde{B}_{g_1,L_1}\right) = Ten\left(B_{g_1,L_1}\right) * C_{L_1}, \tag{3.9}$$

$$Ten\left(\tilde{B}_{g_2,L_2}\right) = Ten\left(B_{g_2,L_2}\right) * C_{L_2}, \tag{3.10}$$

$$Ten\left(B_{g_1,L_1} B_{g_2,L_2}\right) = \frac{1}{C_{L_1+L_2}} * \boldsymbol{Prod}\left(Ten\left(\tilde{B}_{g_1,L_1}\right), Ten\left(\tilde{B}_{g_2,L_2}\right)\right). \tag{3.11}$$

where C_L is the multi-dimensional binomial tensor where its Kth component is equal to

$\binom{L}{K}$, *i.e.,* $(C_L)_K = \binom{L}{K}$. *With some abuse of notation, we use* $1/C_L$ *to denote the multi-dimensional binomial tensor where its* K*th component is equal to* $\frac{1}{\binom{L}{K}}$.

The proof of Proposition 3.3 generalizes the argument in [81] to multi-dimensional inputs and is omitted for brevity. The Bernstein polynomials in (4.16) and (4.17) are called scaled Bernstein polynomials [81] and enjoy the fact that their multiplication corresponds to the multiplication of power series polynomials. Hence we can use the power series **Prod** in (4.19) followed by the element-wise multiplication with the $\frac{1}{C_{L_1+L_2}}$ tensor to remove the effect of the scaling. Recall that we use $A * B$ to denote the element-wise multiplication between the tensors A and B, which can also be carried over using GPUs efficiently which renders all the steps in equations (4.16)-(4.19) to be efficiently implementable on GPUs. We refer to the equations (4.16)-(4.19) as **Prod_Bern**$(B_{g_1,L_1}, B_{g_2,L_2})$.

Using **Prod_Bern**, one can compute the tensor corresponding to raising the function g to power i, where $i \in \mathbb{N}$ is an integer power, denoted by $\text{Ten}(B_{g^i,L})$ by applying the **Prod_Bern** procedure i times. We refer to this procedure as **Pow_Bern**$(\text{Ten}(B_{g,L}), i)$.

3.4.3 Addition between two Bernstein polynomials

The authors in [35] studied how to add two Bernstein polynomials. However, their study is restricted to one-dimensional polynomials which are defined over the unity interval $I_1(x) = [0, 1]$. We extend the argument to the general case with n inputs and any interval $I_n(\underline{d}, \overline{d})$ using the following result.

Proposition 3.4. *Given two Bernstein polynomials* $B_{g_1,L_1}(x)$ *and* $B_{g_2,L_2}(x)$ *with two different orders* $L_1 = (l_1^1, \cdots, l_n^1)$ *and* $L_2 = (l_1^2, \cdots, l_n^2)$. *Define* $L_{sum} = \max(L_1, L_2)$, *where the* max *operator is applied element-wise. The tensor representation of* $B_{g_1+g_2,L_{sum}}$ *can be* *computed as:*

$$L_{sum} = (\max(l_1^1, l_1^2), \ldots, \max(l_n^1, l_n^2)) \tag{3.12}$$

$$Ten\,(B_{g_1,L_{sum}}) = \boldsymbol{Prod\_Bern}\,(Ten\,(B_{g_1,L_1})\,, 1_{L_{sum}-L_1+1}) \tag{3.13}$$

$$Ten\,(B_{g_2,L_{sum}}) = \boldsymbol{Prod\_Bern}\,(Ten\,(B_{g_2,L_2})\,, 1_{L_{sum}-L_2+1}) \tag{3.14}$$

$$Ten\,(B_{g_1+g_2,L_{sum}}) = Ten\,(B_{g_1,L_{sum}}) + Ten\,(B_{g_2,L_{sum}}) \tag{3.15}$$

where 1_{L_e-L+1} *is a multi-dimensional tensor of a shape* $L_e - L + 1$ *that contains just ones.*

The proof of Proposition 4.3 generalizes the argument in [35] and is omitted for brevity. The operation in (4.21) and (4.22) is referred to as *degree elevation* in which we change the dimensions of the tensors ... Once both tensors are of the same dimension, we can add them element-wise. We denote by **Sum_Bern** the procedure defined by (4.20)-(4.23). Again, we note that all the operations in the **Sum_Bern** entail tensor element-wise multiplication and addition

3.5 BERN-NN algorithm

In this section, we provide the details of our tool, named BERN-NN. BERN-NN uses the tensor encoding discussed in Section 4 to propagate Bernstein polynomials that over- and under-approximate the different neurons in the network until over- and under-approximation polynomials for the final output of the network are computed.

3.5.1 Propagating bounds through single neuron

We first discuss how to propagate over- and under-approximations through neurons. Recall our notation that we use $\hat{z}_j^{(i)}$ and $z_j^{(i)}$ to denote the input and output of the j-th neuron in the i-th layer. For ease of notation, we drop the i and j from the notation in this subsection.

Assume that we already computed the over- and under-approximations for the input of one of the hidden neurons, denoted by $\overline{B}_{\hat{z},L_{\hat{z}}}(x)$ and $\underline{B}_{\hat{z},L_{\hat{z}}}(x)$, respectively. The objective is to compute the over- and under-approximations for the output of such a neuron, denoted by $\overline{B}_{z,L_z}(x)$ and $\underline{B}_{z,L_z}(x)$, respectively. We proceed as follows.

Step 1: Compute input bounds for the neuron. Recall that the Bernstein coefficients depend on the input bounds of the function it aims to approximate. Since our aim is to approximate the scalar ReLU function of a neuron, we start by computing the bounds on the input to that neuron as follows:

$$lo = \min_{x \in I_n(\underline{d},\overline{d})} \underline{B}_{\hat{z},L_{\hat{z}}}(x), \qquad hi = \max_{x \in I_n(\underline{d},\overline{d})} \overline{B}_{\hat{z},L_{\hat{z}}}(x) \tag{3.16}$$

Thanks to the enclosure property (3.4), we can solve the optimization problems (3.16) by finding the minimum and the maximum coefficients of $\underline{B}_{\hat{z},L_{\hat{z}}}$ and $\overline{B}_{\hat{z},L_{\hat{z}}}$.

Step 2: Compute the polynomials $\overline{B}_{\sigma,L}$ and $\underline{B}_{\sigma,L}$ that approximate the ReLU function. Given a user-defined approximation order L, the next step is to compute the Bernstein polynomials that over- and under-approximate the ReLU activation function σ denoted by $\overline{B}_{\sigma,L}$ and $\underline{B}_{\sigma,L}$. These polynomials can be computed using the knowledge of lo and hi along with the definition of the Bernstein polynomial in (3.3). To facilitate the computations of the next step, we need to convert these polynomials into the corresponding power series form. This can be done by following the procedure in [78] to obtain:

$$p_{\overline{B}_{\sigma,L}}(x) = \sum_{K \leq L} a_K^{\overline{B}_{\sigma,L}} x^K, \qquad p_{\underline{B}_{\sigma,L}}(x) = \sum_{K \leq L} a_K^{\underline{B}_{\sigma,L}} x^K \tag{3.17}$$

Step 3: Propagate the bounds through the decomposition of polynomials. First, note that the following holds due to the monotonicity of the ReLU function σ and the fact that $z = \sigma(\hat{z})$:

$$\underline{B}_{\hat{z},L_{\hat{z}}}(x) \leq \hat{z}(x) \leq \overline{B}_{\hat{z},L_{\hat{z}}}(x) \Rightarrow \tag{3.18}$$

$$\underline{B}_{z,L_z}(x) \leq \sigma\left(\underline{B}_{\hat{z},L_{\hat{z}}}(x)\right) \leq \underbrace{\sigma\Big(\hat{z}(x)\Big)}_{z(x)} \leq \sigma\left(\overline{B}_{\hat{z},L_{\hat{z}}}(x)\right) \leq \overline{B}_{z,L_z}(x) \tag{3.19}$$

In other words, the post-bounds of the neuron, denoted by $\overline{B}_{z,L_z}(x)$ and $\underline{B}_{z,L_z}(x)$ can be computed by composing the function σ with the under- and over-approximations of the neuron input $\underline{B}_{\hat{z},L_{\hat{z}}}(x)$ and $\overline{B}_{\hat{z},L_{\hat{z}}}(x)$. Indeed such composition is hard to compute due to the nonlinearity in σ. Instead, we perform such composition with the over- and under-approximations of σ, $p_{\overline{B}_{\sigma,L}}$ and $p_{\underline{B}_{\sigma,L}}$, computed in Step 2, as:

$$\underline{B}_{z,L_z}(x) = \sum_{K \leq L} a_K^{\underline{B}_{\sigma,L}} \left(\underline{B}_{\hat{z},L_{\hat{z}}}(x)\right)^K \tag{3.20}$$

$$\overline{B}_{z,L_z}(x) = \sum_{K \leq L} a_K^{\overline{B}_{\sigma,L}} \left(\overline{B}_{\hat{z},L_{\hat{z}}}(x)\right)^K \tag{3.21}$$

Given the tensor representation $Ten(\underline{B}_{\hat{z},L_{\hat{z}}})$ and $Ten(\overline{B}_{\hat{z},L_{\hat{z}}})$, we can use the **Pow_Bern** and **Sum_Bern** procedures to perform the computations in (3.20) and (3.21) to calculate $Ten(\underline{B}_{z,L_z})$ and $Ten(\overline{B}_{z,L_z})$ with $L_z = L_{\hat{z}} * L$.

3.5.2 Propagating the bounds through one layer

Next, we discuss how to propagate the under- and over-approximation polynomials of the outputs of the $i-1$ layer denoted by $\underline{B}_{z_j^{(i-1)},\,L_z}, \overline{B}_{z_j^{(i-1)},\,L_z}, j \in \{1, \ldots, h_{i-1}\}$ to compute under- and over-approximation of the inputs of the neurons in the ith layer $\underline{B}_{\hat{z}_m^{(i)},\,L_{\hat{z}}}, \overline{B}_{z_m^{(i)},\,L_{\hat{z}}}, m \in \{1, \ldots, h_i\}$ of the neural network. Such bound propagation entails composing the under- and over-approximation polynomials $\underline{B}_{z_j^{(i-1)},\,L_z}, \overline{B}_{z_j^{(i-1)},\,L_z}$ with the weights of the ith layer of the neural network $W^{(i)}, b^{(i)}$. To that end, we define the set of positive and negative weights as:

$$W_+^{(i)} = \max\left(W^{(i)}, 0_{i\times(i-1)}\right) \qquad W_-^{(i)} = \min\left(W^{(i)}, 0_{i\times(i-1)}\right).$$

Similarly, for the outputs of the $i-1$ layer of the network, we define the vector of over-approximation polynomials and vector of the under-approximation polynomials as:

$$\overline{B}_{z^{(i-1)},\,L_z} = \left[\overline{B}_{z_1^{(i-1)},L_z} \ldots, \overline{B}_{z_{h_{i-1}}^{(i-1)},L_z}\right]^T,$$
$$\underline{B}_{z^{(i-1)},\,L_z} = \left[\underline{B}_{z_1^{(i-1)},L_z} \ldots, \underline{B}_{z_{h_{i-1}}^{(i-1)},L_z}\right]^T,$$

and for the inputs of the $i$the layer as:

$$\overline{B}_{\hat{z}^{(i)},\,L_{\hat{z}}} = \left[\overline{B}_{\hat{z}_1^{(i)},L_{\hat{z}}} \ldots, \overline{B}_{\hat{z}_{h_i}^{(i)},L_{\hat{z}}}\right]^T$$
$$\underline{B}_{\hat{z}^{(i)},\,L_{\hat{z}}} = \left[\underline{B}_{\hat{z}_1^{(i)},L_{\hat{z}}} \ldots, \underline{B}_{\hat{z}_{h_i}^{(i)},L_{\hat{z}}}\right]^T$$

Hence, the over- and under-approximations of the inputs of the ith layer can be efficiently computed as:

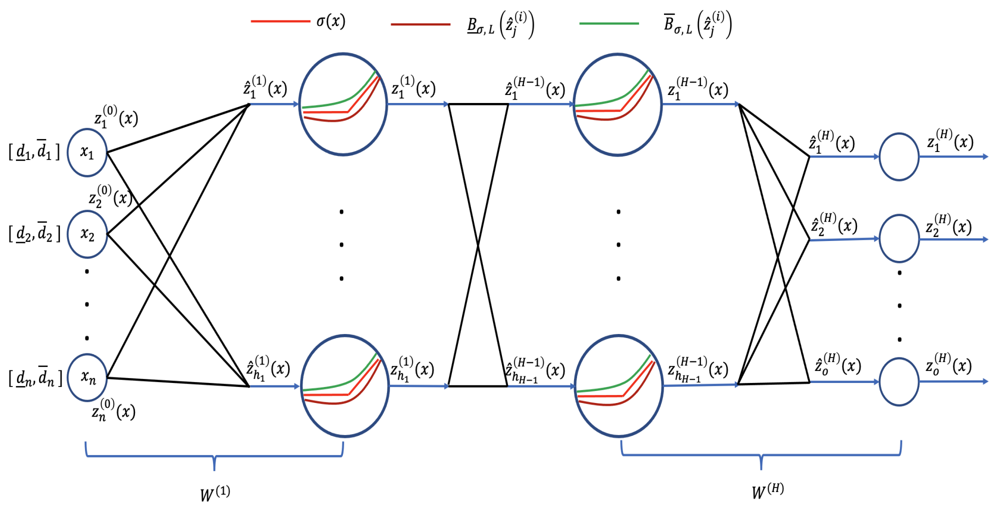


Figure 3.3: Mechanism of BERN-NN Polynomial Interval Arithmetic.

$$Ten\left(\overline{B}_{\hat{z}^{(i)},L_{\hat{z}}}\right) = W_+^{(i)} \times Ten\left(\overline{B}_{z^{(i-1)},L_z}\right) + W_-^{(i)} \times Ten\left(\underline{B}_{z^{(i-1)},L_z}\right) + b^{(i)} \tag{3.22}$$

$$Ten\left(\underline{B}_{\hat{z}^{(i)},L_{\hat{z}}}\right) = W_+^{(i)} \times Ten\left(\underline{B}_{z^{(i-1)},L_z}\right) + W_-^{(i)} \times Ten\left(\overline{B}_{z^{(i-1)},L_z}\right) + b^{(i)} \tag{3.23}$$

3.5.3 Mechanism of BERN-NN Polynomial Interval Arithmetic

We finally describe the proposed BERN-NN Polynomial Interval Arithmetic algorithm, depicted in Figure 3.3. For a neural network with n inputs $x_1, \ldots, x_n$, we initialize an over- and under-approximation Bernstein polynomials for each of the inputs, i.e.,:

$$\overline{B}_{z_i^{(0)},1} = \underline{B}_{z_i^{(0)},1} = \overline{B}_{z_i^{(0)},1} \qquad i \in \{1, \ldots, n\}.$$

Note that in the equation above, we used $z_i^{(0)}$ as a replacement of x_i to unify the notation with the remainder of the operations (see Figure 3.3). To compute the Bernstein polynomials $\overline{B}_{z_i^{(0)},1}$ and $\underline{B}_{z_i^{(0)},1}$, we recall that the coefficients of such polynomials depend on the input domain. Hence, given a hypercube $I_n(\underline{d}, \overline{d})$ that bounds the input x of the neural network, we compute the tensor representation of these polynomials as:

$$Ten\left(\overline{B}_{z_1^{(0)},1}\right) = Ten\left(\underline{B}_{z_1^{(0)},1}\right) = \begin{bmatrix} \underline{d}_1 \\ \overline{d}_1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \ldots \otimes \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{3.24}$$

$$Ten\left(\overline{B}_{z_2^{(0)},1}\right) = Ten\left(\underline{B}_{z_2^{(0)},1}\right) = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} \underline{d}_2 \\ \overline{d}_2 \end{bmatrix} \otimes \ldots \otimes \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{3.25}$$

$$\vdots$$

$$Ten\left(\overline{B}_{z_n^{(0)},1}\right) = Ten\left(\underline{B}_{z_n^{(0)},1}\right) = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \ldots \otimes \begin{bmatrix} \underline{d}_n \\ \overline{d}_n \end{bmatrix} \tag{3.26}$$

Next, we propagate these over- and under-approximation polynomials to the inputs of the first layer in the neural network using (4.2) and (4.3). Given a user-defined approximation order L, we propagate the polynomial approximations through the ReLU function using (3.20) and (3.21) for each of the neurons in layer 1. The produced over- and under-approximations of the outputs of all neurons are aggregated together in one tensor which is then propagated to the next layer. This process continues until we compute the over- and under-approximation polynomials of the outputs of the neural network, denoted by $\overline{B}_{z_j^{(H)},L^{H-1}}(x), \underline{B}_{z_j^{(H)},L^{H-1}}(x)$ for $j = 1, \ldots, o$. These polynomials are used as the solution of Problem 1.

It is important to note that the final Bernstein polynomials $\overline{B}_{z_j^{(H)},L^{H-1}}(x), \underline{B}_{z_j^{(H)},L^{H-1}}(x)$ have orders of L^{H-1} where L is the user-defined order of approximation of the ReLU function and H is the number of layers. This polynomial order increases exponentially with the number of hidden layers. Similarly, the shape of their multi-dimensional tensor representations is equal to $L^{H-1}+1$ which increases exponentially with the number of hidden layers. To alleviate this problem, we introduce a parameter called *Lin*. Based on this parameter, we drop the orders of the post-bound over- and under-approximation polynomials to $[1, \cdots, 1]$. In other words, we linearize the approximation polynomials every *Lin* hidden layers. We use the algorithm in [47] to perform such linearization of the Bernstein polynomial. Luckily, this algorithm, like all the other operations in our BERN-NN involves tensor multiplications and additions and hence can be parallelized over GPUs efficiently.

Finally, note that one can always obtain absolute bounds on the inputs or outputs of any of the neurons (including the outputs of the neural network), thanks to the enclosure property of Bernstein polynomials (3.4). Such absolute bounds are useful for reachability analysis and model checkers.

3.5.4 GPU Implementation Details

To get the performance increase of GPUs without the complications of low-level languages, we implemented this tool in PyTorch. As mentioned above, we represent n-dimensional Bernstein polynomials as dense n-dimensional tensors. The tool becomes memory bound very quickly as the number of input nodes increases, making the number of dimensions in the tensors larger. In order to combat this, we use as many in-place operations as possible to avoid repeatedly allocating large chunks of memory during computation. Similarly, the multinomial coefficients used for degree elevation are used multiple times throughout the tool, and we cache each the first time they are generated to avoid spending time re-doing calculations and allocating additional memory.

We parallelized the tool on a node level: at each layer, the outputs of the last layer are passed to each node, which then can run independently of each other on separate GPUs. However, because the tensors become large very quickly, the gains in computation time only offset the overhead of copying tensors between GPUs when the neural network is particularly large. We collect and stack the outputs of all the nodes in one tensor and pass it to the next layer. When the polynomials are being composed with the ReLU approximation, each term is elevated to the highest degree expected of a composition between these two polynomials.

This both ensures that the outputs of all the neurons can be stacked, as they are all the same shape and size, and also allows the multiplication of the stacked outputs of the last layer by the incoming weights to be a simple broadcasting multiplication, which is then easily parallelizable on a GPU.

We achieved additional performance gains by rewriting for-loops as element-wise tensor operations and by batching linear algebra operations like matrix multiplications and calculating the least-square solutions of matrices, both of which allow operations to be easily parallelized on GPUs and reduce the amount of time spent allocating many small patches of memory, instead doing a single large allocation.

3.6 Numerical Results

In this section, we perform a series of numerical experiments to evaluate the scalability and effectiveness of our tool. First, we conduct an ablation study to check the effect of varying different parameters (e.g., neural network width, neural network depth, ReLU approximation order) on the performance of our tool. We utilize two metrics:

- **Execution time**: which measures the time (in seconds) needed to compute the final Bernstein polynomials. Indeed, smaller values indicate better performance.
- **Relative volume of the output set**: this metric measures the "tightness" of the produced over- and under-approximation polynomials. Without loss of generality, we

focus on neural networks with one output $z^{(H)}$ and we compute this metric as:

$$\text{Vol\_relative} = \frac{\text{Vol\_Output}}{\text{Vol\_Input}} \tag{3.27}$$

$$\text{Vol\_Input} = \prod_{i=1}^{n} \left(\overline{d}_i - \underline{d}_i \right) \tag{3.28}$$

$$\text{Vol\_Output} = \int \cdots \int_{I_n} \left(\overline{B}_{z^{(H)}}(x) - \underline{B}_{z^{(H)}}(x) \right) dx_1 \ldots dx_n \tag{3.29}$$

Indeed, smaller values of this metric indicate tighter approximations of the output set.

After the ablation study, we compare our tool with a set of state-of-the-art bound computation tools—including the winner of the last 2022 Verification of Neural Network (VNN) competition [5]—to study the relative performance.

Setup: We implemented our tool in Python3.9 using PyTorch for all tensor arithmetic. We run all our experiments using a single GeForce RTX 2080 Ti GPU and two 24-core Intel(R) Xeon(R). We like to note that the throughput of the tool can be increased by utilizing multiple GPU to process different neurons in parallel in a batch-processing fashion. However, in this section, we focus on using only one GPU and we leave the generalization of our algorithm to utilize multiple GPUs for future work.

3.6.1 Ablation study

The effect of varying the ReLU's order of approximation:

We study the effect of varying the ReLU's order of approximation L for a fixed NN architecture on the execution time and the output's relative volume space of our tool. In Figure 3.4, we report the statistical results for 50 random networks of a fixed architecture. Figure 3.4 (top) shows that increasing the approximation order increases the execution time. On

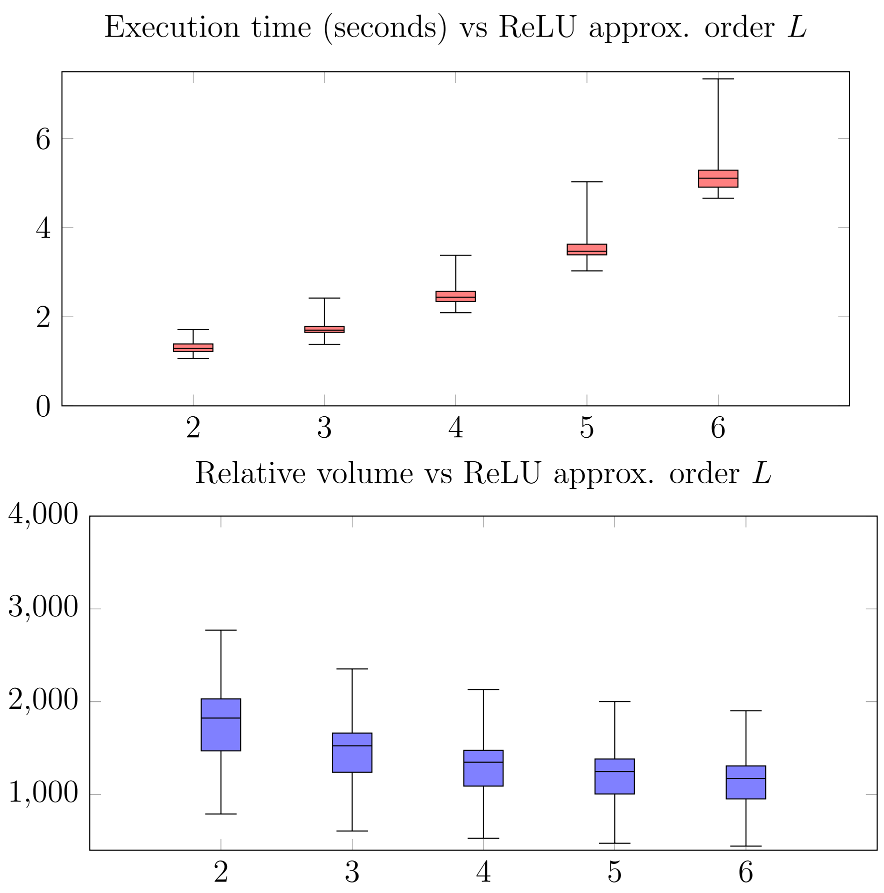


Figure 3.4: Effect of varying the ReLU's order of approximation L for a NN architecture $[2, 20, 20, 1]$ on the execution time of our tool (top) and the relative volume of the output set (bottom). We set $n = 2$, $I_n = [-1, 1]^n$, and $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. The reported results are generated for 50 experiments.

the other hand, Figure 3.4 (bottom) shows that the relative volume of the output set significantly decreases with increasing the order of approximation. The results of both figures highlight the trade-off between the tightness of the output bounds and the execution time as a function of the ReLU approximation order L.

The effect of varying the input's dimension:

We study the effect of varying the input's dimension n, for a fixed NN architecture on the execution time of our tool. Figure 3.5 shows that the execution time for computing the

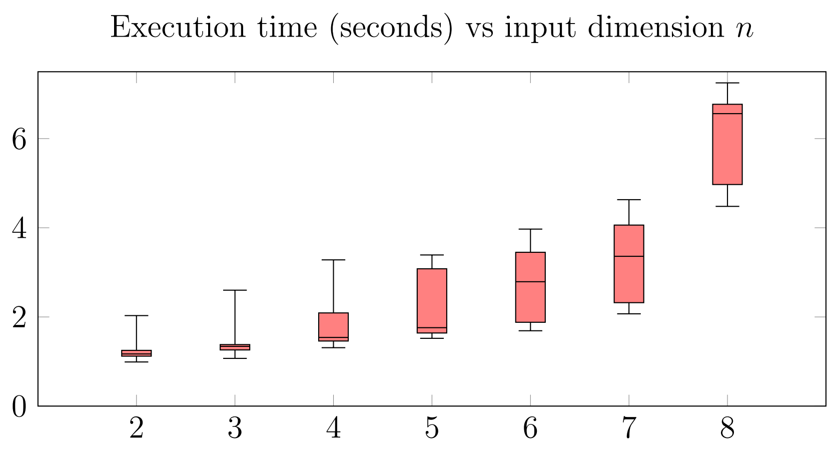


Figure 3.5: Effect of varying the input's dimension n for a NN architecture $[n, 20, 20, 1]$ on the execution time our tool. We set $L = 2$, $I_n = [-1, 1]^n$, and $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. The reported results are generated for 50 experiments.

output set grows linearly for smaller values of n but seems to grow more rapidly after $n = 7$. This suggests that the proposed tool can be used efficiently for many control applications.

The effect of increasing the number of neurons per layer:

We study the effect of varying the number of neurons per layer N_e, for a fixed NN architecture $[3, N_e, N_e, 1]$ on the execution time of our tool. Figure 3.6 summarizes the execution times with a varying number of neurons per layer. The results show that increasing the number of neurons per layer highly affects the execution time. This is due to the expensive arithmetic and memory operations for large tensors that represent the Bernstein polynomials. Nevertheless, this increase in execution time can be harnessed by using multiple GPUs to compute bounds for different nodes in parallel along with using the same GPU to process multiple nodes simultaneously.

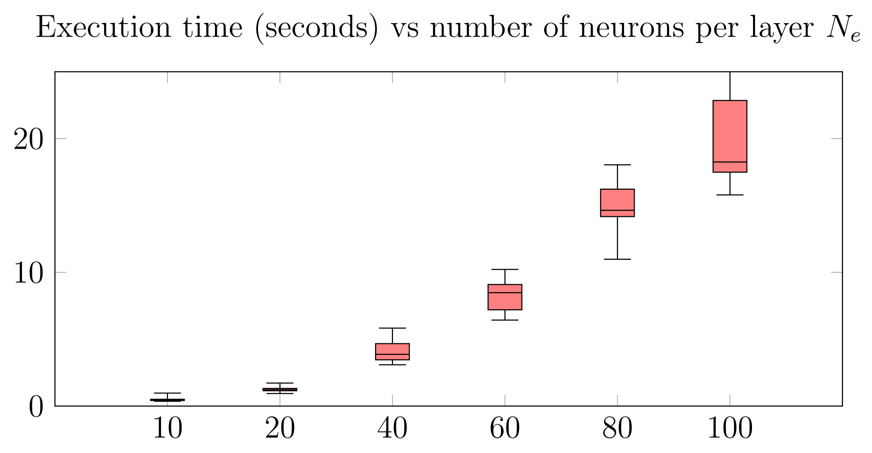


Figure 3.6: Effect of varying the number of neurons per layer N_e for a NN architecture $[2, N_e, N_e, 1]$ on the execution time of our tool. We set $n = 2$, $L = 2$, $I_n = [-1, 1]^n$, and $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. The reported results are generated for 50 experiments.

The effect of increasing the number of hidden layers:

We study the effect of varying the number of hidden layers n_h, with 20 neurons in every hidden layer, on the execution time of our tool. Unlike the effect of increasing the number of neurons per layer, the results in Figure 3.7 show that the execution time almost grows linearly with the number of hidden layers.

Scalability analysis of Bern-NN:

We finally try to study the execution time of Bern-NN for relatively large neural networks. In this study, we add extra layers with 100 neurons each and report the execution time in Figure 3.8 for random neural networks. As shown in the figure, Bern-NN can process neural networks with more than 1000 neurons in less than 2 minutes.

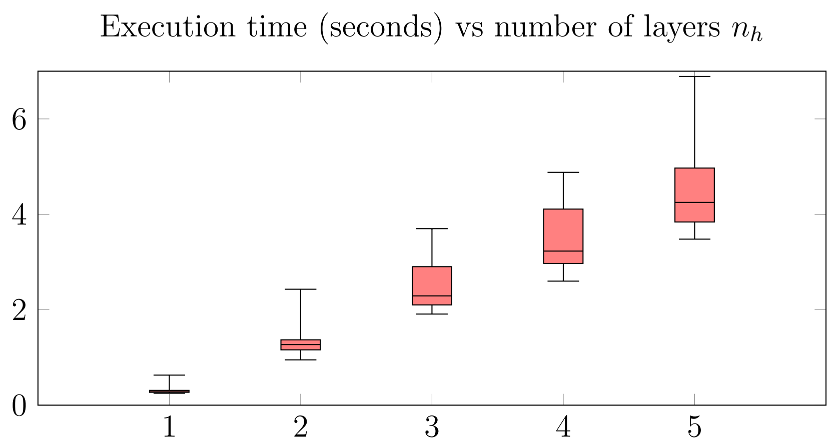


Figure 3.7: Effect of varying the number of hidden layers n_h, for a NN architecture $[2, 20, .., 20, 1]$ with 20 neurons in every hidden layer on the execution time of our tool. We set $n = 2$, $L = 2$, $I_n = [-1, 1]^n$, and $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. The reported results are generated for 50 experiments.

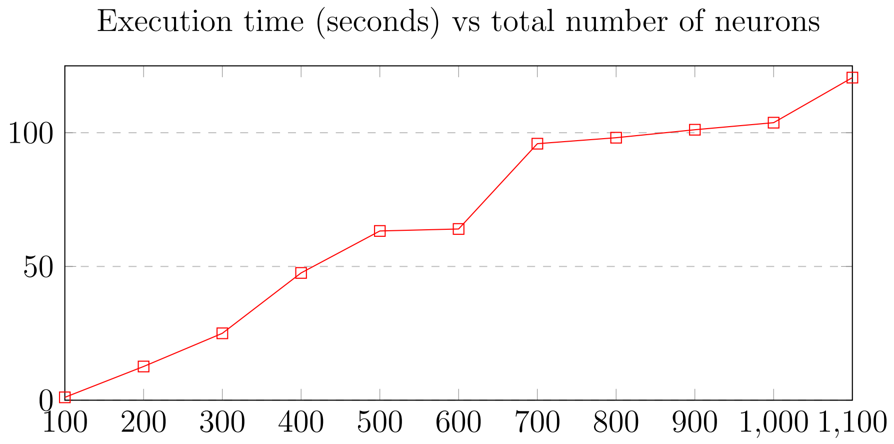


Figure 3.8: Scalability of the Bern-NN tool as a function of increasing the total number of neurons.

3.6.2 Comparison against other tools

In this subsection, we compare the performance of our tool in terms of execution time and the output set's relative volume compared to bound propagation tools such as Symbolic Interval Analysis (SIA)[96], alpha-CROWN [103], and reachability analysis tool such as POLAR [53]. We note that alpha-CROWN [103] was the winner of the 2022 VNN competition and we compare Bern-NN against the bound propagation algorithm used within alpha-CROWN as a representative tool for all the bound propagation techniques. Moreover, alpha-CROWN is also designed to harness the computational powers of GPUs. We compare Bern-NN against POLAR since it also uses polynomials (Taylor Model with a Bernstein error correction) to compute bounds on the output of neural networks. POLAR [53] outperforms other reachability-based tools and hence is a representative tool for such techniques.

Comparison against SIA and alpha-CROWN for random NN

We compare the performance of our tool to SIA and alpha-CROWN for random neural networks with $[2, 20, 20, 1]$ architecture for different hyperrectangle input spaces (Figure 3.9). We also compare the performance as the input dimension of the network increases (Figure 3.10). The results show that SIA is the fastest in terms of execution time for all different input hyperrectangles due to the simplicity of its computations. However, its relative volume is the highest. On the other hand, Bern-NN's relative volume is the smallest for all different input spaces thanks to its tight higher-order ReLU approximations. Compared to alpha-CROWN (which also runs on GPUs), Bern-NN is both faster and produces tighter bounds leading to an average of 25% reduction in execution time with an average of 10% reduction in the relative volume metric. This shows the practicality of Bern-NN for control applications.

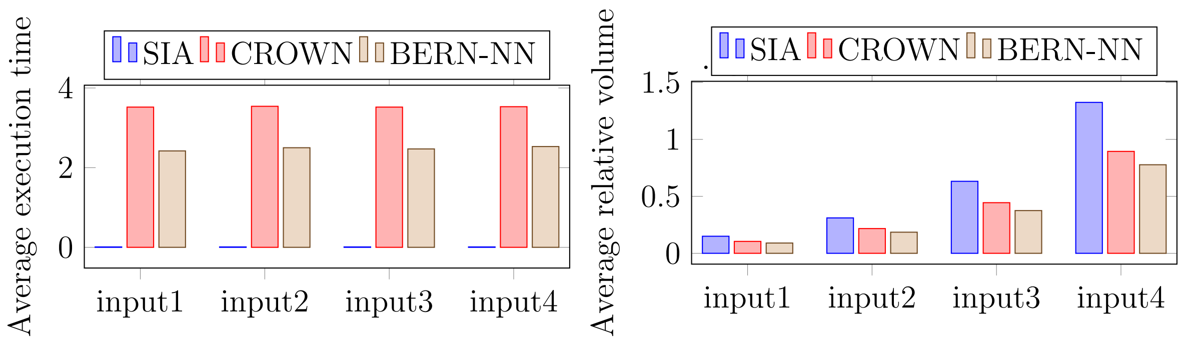


Figure 3.9: Performance results in terms of average execution times (sec) (left) and relative volume (right) for BERN-NN, SIA, and alpha-CROWN for different input spaces. The NN's architecture is $[2, 20, 20, 1]$. The ReLU's order of approximation is $L = 4$, and $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. Input1 $= I_n = [-5, 5]^2$, Input2 $= I_n = [-10, 10]^2$, Input3 $= I_n = [-20, 20]^2$, Input4 $= I_n = [-40, 40]^2$.

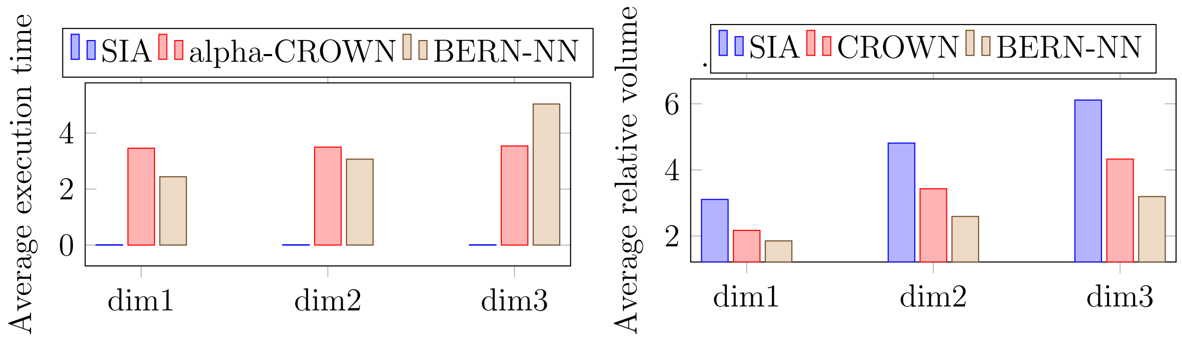


Figure 3.10: Performance results in terms of average execution times (sec) (left) and relative volume (right) for BERN-NN, SIA, and alpha-CROWN for input's dimensions n. The NN's architecture is $[n, 20, 20, 1]$. the input's space is $[-10, 10]^n$. The ReLU's order of approximation is $L = 4$, $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. dim1 $= n = 2$, dim2 $= n = 3$, dim3 $= n = 4$.

Table 3.2: Performance results in terms of average execution times and volume for BERN-NN, SIA, alpha-CROWN, and POLAR, for 5 different input's spaces $I_n\left(\underline{d}, \overline{d}\right)$ for 6 benchmarks [53]. The ReLU's order of approximation is $L = 2$, $Lin = 0$.

Tool	Benchmark 1		Benchmark 2		Benchmark 3		Benchmark 4		Benchmark 5		Benchmark 6	
	[2,20,20,1]		[2,20,20,1]		[2,20,20,1]		[3,20,20,1]		[3,100,100,1]		[4,20,20,20,1]	
	time	volume	time	volume	time	volume	time	volume	time	volume	time	volume
SIA	**0.01**	2.544	**0.02**	6.05	**0.01**	1.02	**0.01**	9.41	**0.02**	53.38	**0.02**	2.03
$CROWN$	2.9	3.1	3.49	5.50	3.54	**0.73**	3.13	17.04	3.80	77.72	4.10	2.4
$Bern-NN$	0.84	**1.62**	1.30	**5.4**	1.09	0.81	1.15	**6.21**	41.7	**35.85**	3.25	**1.38**
$POLAR$	0.21	25.43	0.284	51.80	0.29	18.81	0.42	33.32	5.52	432.75	0.81	7.00

Case Study for Control Benchmarks

In this experiment, we test different tools on benchmarks of NN controllers (used by POLAR) to evaluate the tightness of their estimated bounds. POLAR and BERN-NN use different Bernstein polynomials to approximate ReLU functions. Proposition 2 shows the maximum error of ReLU's Bernstein overapproximation equals the Bernstein approximation at 0. This error is the smallest for ReLU's Bernstein overapproximation. However, POLAR approximates the ReLU using samples and adds a symmetric error bound estimated using the Lipschitz constant of ReLU functions. Such a method is conservative and leads to additional errors for each ReLU function. Table 4.2 represents six standard benchmarks used by the authors in [53] to evaluate the POLAR tool. Every benchmark represents a trained NN controller for a closed-loop system. Numbers in the square brackets refer to NN architecture, e.g., [2,20,20,1] means the NN has an input layer of two neurons, two hidden layers of 20 neurons, and an output layer of 1 neuron. Table 4.2 summarizes the performance of the tools with respect to the average execution time and average relative volume for six control benchmarks. The results show that Bern-NN provides the tightest estimate for the output set for all benchmarks except Benchmark 3. We would like to highlight that the tight approximation provided by Bern-NN is important for control applications because the specification of interest is usually defined over a time horizon and require multi-step reachability, hence, tighter bounds at each step are crucial. Lastly, Bern-NN is faster than alpha-CROWN over all benchmarks except Benchmark 5. However, SIA and POLAR are faster than Bern-NN but provide looser bound estimates. Each benchmark is run with five different hyperrectangles that are all centered around zero and have a radius $r \in \{1, 1.5, 2, 2.5, 3\}$.

Chapter 4

BERN-NN-IBF: Enhancing Neural Network Bound Propagation Through Implicit Bernstein Form and Optimized Tensor Operations

In this chapter, we present a significant enhancement of the Bernstein-polynomial-based bound propagation algorithm BERN-NN that it is introduced in chapter 3. BERN-NN-IBF offers three main contributions: (i) a memory-efficient encoding of Bernstein-polynomials to scale the bound propagation algorithms, (ii) optimized tensor operations for the new polynomial encoding to maintain the integrity of the bounds while enhancing computational efficiency, and (iii) tighter under-approximations of the ReLU activation function using quadratic polynomials tailored to minimize approximation errors. Through comprehensive testing, we demonstrate that BERN-NN-IBF achieves tighter bounds and higher computational efficiency compared to the original BERN-NN and state-of-the-art methods, including linear programming and convex used within the winner of the VNN-COMPETITION.

4.1 Introduction

As neural networks (NNs) take the driver's seat in powering critical cyber-physical systems, the urgency to validate their reliability and safety escalates [89, 90, 43]. From steering autonomous vehicles to orchestrating smart cities technologies and managing complex avionic systems, these networks operate at the heart of applications where error margins must be near non-existent. The intricate task of NN verification hinges on precise bound propagation - a process challenged by the networks' non-linear and non-convex nature, making exact output bound determination an NP-hard problem [57].

Previous methodologies leaned heavily on either linear relaxation techniques [27, 14, 91, 8, 17, 40, 3, 20] or reachability analysis [102, 101, 49, 97, 92, 55, 37], both of which often culminate in substantial over-approximations. BERN-NN (chapter 3) [36], took a detour from a traditional practices by harnessing Bernstein polynomials' power for more accurate approximations of non-linear activations. While BERN-NN marked a significant leap forward, the quest for enhanced precision and efficiency remained ongoing.

In this chapter, we introduce BERN-NN-IBF, an improvement tool of BERN-NN. BERN-NN-IBF introduces the implicit Bernstein form (IBF), a novel representation that employs 2D tensors for bounds, sidestepping the computational hurdles synonymous with the n-dimensional tensors used in BERN-NN.

Moreover, we have designed new operations specific to the IBF format, enhancing essential functions like summations, multiplications, and power, which are implemented efficiently using specialized CUDA routines, contributing significantly to the tool's overall speed.

Finally, we proposed in BERN-NN-IBF an approach for calculating the coefficients of a quadratic polynomial that under-approximates the ReLU function more accurately. We achieve the lowest approximation error by formulating and solving a dedicated optimization

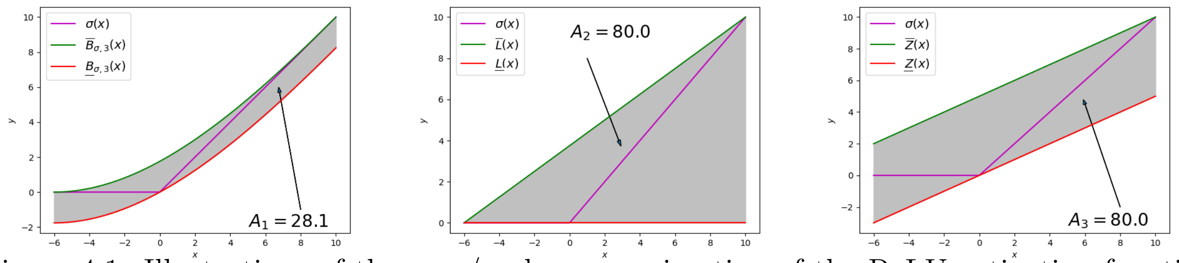


Figure 4.1: Illustrations of the over/under-approximation of the ReLU activation functions in the interval $\big[-6, 10\big]$ using different approaches: Higher-order polynomials (Left), triangulation (Center), and zonotope (Right). The area of the shaded set A_1, A_2, A_3 represents the approximation error for each of the approaches (chapter 3) [36].

problem, ensuring even tighter bounds on the NN's outputs. This sophisticated method surpasses the Bernstein approximation used in BERN-NN.

Our extensive evaluations confirm that BERN-NN-IBF does not just outpace its predecessor BERN-NN but also sets new benchmarks in tight bound attainment, overshadowing traditional methods and even acclaimed tools like those used in the winner of the VNN-COMPETITON (alpha-beta-CROWN [99]).

4.2 Neural Network Bound Propagation Using Bernstein Polynomials

In this chapter, we seek to find higher-order polynomials that upper and lower approximate the outputs of a ReLU-based Neural Network $NN(x)$ whenever the NN's input x is confined within a pre-defined hypercube, i.e., $x \in I_n(\underline{d}, \overline{d})$. As shown in chapter 3 [36] and visualized in Figure 4.1, higher-order polynomial approximation of ReLU functions outperform state-of-the-art approaches of using linear approximations (e.g., triangulation, crown, and zonotopes) in terms of approximation errors. Albeit promising, the propagation of bounds encoded as higher-order polynomials across different NN layers is computationally challenging compared to linear approximations. In this section, we review the basics of higher-order polynomial approximation of ReLU functions and discuss their challenges.

General notation: We use the symbols $\mathbb{N}$ and $\mathbb{R}$ to denote the set of natural and real numbers, respectively. We denote by $x = \big(x_1, x_2, \cdots, x_n\big) \in \mathbb{R}^n$ the vector of n real-valued variables, where $x_i \in \mathbb{R}$. We denote by $I_n(\underline{d}, \overline{d}) = \big[\underline{d}_1, \overline{d}_1\big] \times \cdots \times \big[\underline{d}_n, \overline{d}_n\big] \subset \mathbb{R}^n$ the n-dimensional hyperrectangle where $\underline{d} = (\underline{d}_1, \cdots, \underline{d}_n)$ and $\overline{d} = \big(\overline{d}_1, \cdots, \overline{d}_n\big)$ are the lower and upper bounds of the hyperrectangle, respectively.

For a real-valued vector $x = \big(x_1, x_2, \cdots, x_n\big) \in \mathbb{R}^n$ and an index-vector $K = (k_1, \cdots, k_n) \in \mathbb{N}^n$, we denote by $x^K \in \mathbb{R}$ the scalar $x^K = x_1^{k_1} \times \ldots \times x_n^{k_n}$. Given two multi-indices $K = (k_1, \cdots, k_n) \in \mathbb{N}^n$ and $L = (l_1, \cdots, l_n) \in \mathbb{N}^n$, we use the following notation throughout this paper:

$$\begin{aligned} K + L &= (k_1 + l_1, \cdots, k_n + l_n), \\ \binom{L}{K} &= \binom{l_1}{k_1} \times \cdots \times \binom{l_n}{k_n}, \\ \sum_{K \le L} &= \sum_{k_1 \le l_1} \cdots \sum_{k_n \le l_n} \end{aligned}$$

Finally, a real-valued multivariate polynomial $p : \mathbb{R}^n \to \mathbb{R}$ is defined as:

$$\begin{aligned} p(x_1, \ldots, x_n) &= \sum_{k_1=0}^{l_1} \sum_{k_2=0}^{l_2} \cdots \sum_{k_n=0}^{l_n} a_{(k_1,\ldots,k_n)} x_1^{k_1} x_2^{k_2} \ldots x_n^{k_n} \\ &= \sum_{K \le L} a_K x^K, \end{aligned}$$

where $L = (l_1, l_2, \ldots, l_n)$ is the maximum degree of x_i for all $i = 1, \ldots, n$.

4.2.1 Bernstein Polynomials:

Definition 4.1. *(Bernstein Polynomials) Consider a continuous polynomial $p : \mathbb{R}^n \to \mathbb{R}$, defined over an input domain (hypercube) $I_n(\underline{d}, \overline{d}) \subseteq \mathbb{R}^n$, and a multi-index $L = (l_1, \ldots, l_n) \in \mathbb{N}^n$. The Lth order Bernstein polynomial of p is given by:*

$$\begin{aligned} B_{p,L}(x) &= \sum_{K \le L} b^p_{K,L} \cdot Ber_{K,L}(x), \\ Ber_{K,L}(x) &= \binom{L}{K} \cdot \frac{(x-\underline{d})^K (\overline{d}-x)^{L-K}}{(\overline{d}-\underline{d})^L}, \end{aligned} \tag{4.1}$$

where $Ber_{K,L}(x)$ denotes the Bernstein basis and $b^p_{K,L}$ represents the Bernstein coefficients of p.

Bernstein polynomials are particularly noted for their capacity to approximate any continuous function on a closed interval. This property is crucial when dealing with functions that are not differentiable, as is the case with certain activation functions in neural networks, such as the Rectified Linear Unit (ReLU).

4.2.2 Interval Bound Propagation Using Bernstein Polynomials

The BERN-NN framework employs Bernstein polynomials for accurately propagating interval bounds through neural networks (NNs). Illustrated in Figure 4.2, a simple NN example with two input neurons, a hidden layer of two neurons, and a single output neuron demonstrates the process. The NN operates within an input domain $x \in I_2(\underline{d}, \overline{d})$, with specified weights $W = [W^{(1)}, W^{(2)}]$ and biases $b = [b^{(1)}, b^{(2)}]$. Initially, BERN-NN calculates lower and upper Bernstein bounds for the input, designated as $\begin{bmatrix} Den(\overline{B}_i) \\ Den(\underline{B}_i) \end{bmatrix}$ for $i = 0, 1$. These bounds are then propagated to the hidden layer using positive and negative weight partitions, $W^{(1)}_+$

and $W_{-}^{(1)}$, according to:

$$\begin{bmatrix} Den\left(\overline{B}_2\right) \\ Den\left(\overline{B}_3\right) \end{bmatrix} = W_{+}^{(1)} \times \begin{bmatrix} Den\left(\overline{B}_0\right) \\ Den\left(\overline{B}_1\right) \end{bmatrix} + W_{-}^{(1)} \times \begin{bmatrix} Den\left(\underline{B}_0\right) \\ Den\left(\underline{B}_1\right) \end{bmatrix} + b^{(1)} \tag{4.2}$$

$$\begin{bmatrix} Den\left(\underline{B}_2\right) \\ Den\left(\underline{B}_3\right) \end{bmatrix} = W_{+}^{(1)} \times \begin{bmatrix} Den\left(\underline{B}_0\right) \\ Den\left(\underline{B}_1\right) \end{bmatrix} + W_{-}^{(1)} \times \begin{bmatrix} Den\left(\overline{B}_0\right) \\ Den\left(\overline{B}_1\right) \end{bmatrix} + b^{(1)} \tag{4.3}$$

where

$$W_{+}^{(1)} = \max\left(W^{(i)}, 0_{i\times(i-1)}\right), W_{-}^{(i)} = \min\left(W^{(i)}, 0_{i\times(i-1)}\right)$$

denote the set of positive and negative weights between the $(i-1)$th layer to the ith layer.

Next, BERN-NN over/under-approximate the ReLU σ using Bernstein polynomials $\overline{B}_{\sigma,L}$ and $\underline{B}_{\sigma,L}$. After that, it propagates these bounds through the ReLU bounds as follows:

$$Den\left(\overline{B}_4\right) = \overline{B}_{\sigma,L}\left(Den\left(\overline{B}_2\right)\right) \tag{4.4}$$

$$Den\left(\overline{B}_5\right) = \overline{B}_{\sigma,L}\left(Den\left(\overline{B}_3\right)\right) \tag{4.5}$$

$$Den\left(\underline{B}_4\right) = \underline{B}_{\sigma,L}\left(Den\left(\underline{B}_2\right)\right) \tag{4.6}$$

$$Den\left(\underline{B}_5\right) = \underline{B}_{\sigma,L}\left(Den\left(\underline{B}_3\right)\right) \tag{4.7}$$

Finally, BERN-NN propagates the resultant bounds to the output neuron. These final Bernstein bounds represent bound the NN's actual output:

$$\begin{bmatrix} Den\left(\overline{B}_6\right) \end{bmatrix} = W_+^{(2)} \times \begin{bmatrix} Den\left(\overline{B}_4\right) \\ Den\left(\overline{B}_5\right) \end{bmatrix} + W_-^{(2)} \times \begin{bmatrix} Den\left(\underline{B}_4\right) \\ Den\left(\underline{B}_5\right) \end{bmatrix} + b^{(2)} \tag{4.8}$$

$$\begin{bmatrix} Den\left(\underline{B}_6\right) \end{bmatrix} = W_+^{(2)} \times \begin{bmatrix} Den\left(\underline{B}_4\right) \\ Den\left(\underline{B}_5\right) \end{bmatrix} + W_-^{(2)} \times \begin{bmatrix} Den\left(\overline{B}_4\right) \\ Den\left(\overline{B}_5\right) \end{bmatrix} + b^{(2)} \tag{4.9}$$

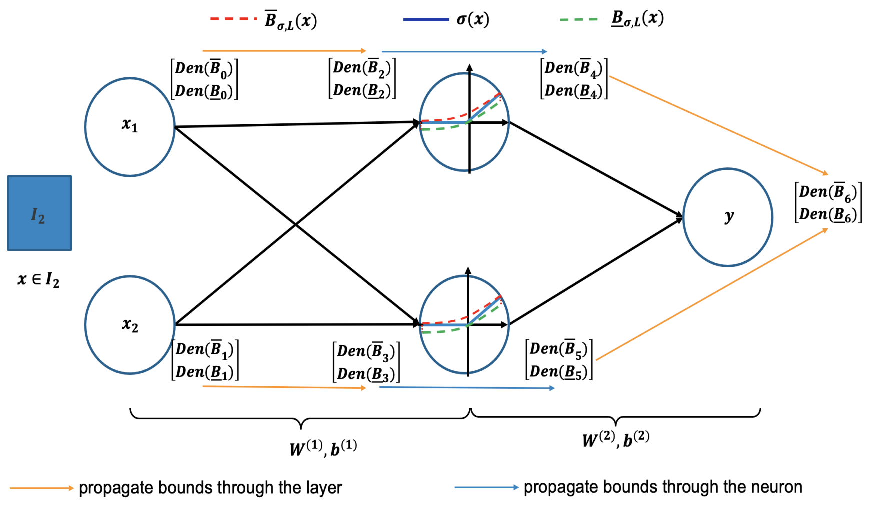
BERN-NN uses two algorithms for the summation **Sum_Bern** and multiplication **Prod_Bern** to perform these computations over the dense representation of Bernstein polynomials (chapter 3) [36]. Algorithm 7 represents the BERN-NN algorithm (chapter 3) [36].

In the following proposition, we show how the dense representation of Bernstein polynomial scales with respect to the dimension n and order L.

Proposition 4.1. *Given n-dimenstional Bernstein polynomial of order* $L = (l_1, \cdots, l_n)$, $B_{p,L}(x)$. *Its dense representation* $Den\left(B_{p,L}\right)$ *is* $n-$*dimensional tensor of a size* $(l_1+1, \cdots, l_n+1)$ *(chapter 3) [36]. The total components of the dense representation is* $O\left(l_{max}^n\right)$, *where* $l_{max} = \max\limits_{1\leq i\leq n} l_i$.

Proof. The total components of the dense representation is $\prod\limits_{i=1}^{n}(l_i+1)$ (chapter 3) [36]. By picking $l_{max} = \max\limits_{1\leq i\leq n} l_i$, we get the results. □

Proposition 4.1 shows that the dense representation scales exponentially with the dimension n which makes BERN-NN inconvenient for higher-dimensional input space. In the next section, we introduce a new representation called Implicit representation that circumvent this problem.



$\overline{B}_{\sigma,L}(x)$
$\sigma(x)$
$\underline{B}_{\sigma,L}(x)$
I_2
$x \in I_2$
x_1
x_2
y
$\begin{bmatrix} Den(\overline{B}_0) \\ Den(\underline{B}_0) \end{bmatrix}$
$\begin{bmatrix} Den(\overline{B}_1) \\ Den(\underline{B}_1) \end{bmatrix}$
$\begin{bmatrix} Den(\overline{B}_2) \\ Den(\underline{B}_2) \end{bmatrix}$
$\begin{bmatrix} Den(\overline{B}_3) \\ Den(\underline{B}_3) \end{bmatrix}$
$\begin{bmatrix} Den(\overline{B}_4) \\ Den(\underline{B}_4) \end{bmatrix}$
$\begin{bmatrix} Den(\overline{B}_5) \\ Den(\underline{B}_5) \end{bmatrix}$
$\begin{bmatrix} Den(\overline{B}_6) \\ Den(\underline{B}_6) \end{bmatrix}$
$W^{(1)}, b^{(1)}$
$W^{(2)}, b^{(2)}$
propagate bounds through the layer
propagate bounds through the neuron

Figure 4.2: BERN-NN propagates the interval bounds using the dense representation of Bernstein polynomials (chapter 3) [36].

Algorithm 7 BERN-NN Algorithm for Neural Network Bound Propagation

Input: Neural network input space $I_n\left(\underline{d}, \overline{d}\right)$, approximation order L, NN's weights and biases W and b.
Output: Over- and under-approximation Bernstein polynomials of network outputs.

```
1: Compute over- and under-approximation Bernstein polynomials for network inputs
2: for each layer $i$ from 1 to $H$ do
3:     for each neuron $j$ in layer $i$ do
4:         Propagate bounds through every $j$th neuron in the $i$th layer using Eqs. 3-7.
5:     end for
6:     Propagate bounds to next layer using Eqs 2-3.
7: end for
```

4.3 Memory-Efficient Representation Of Bernstein Polynomials

Building upon the dense representation discussed earlier, we now explore the implicit representation, which offers a more memory-efficient approach, particularly beneficial for high-dimensional scenarios. Given a multivariate polynomial p of order $L = (l_1, \cdots, l_n)$, that consists of t terms, as follows:

$$p\left(x_1, \cdots, x_n\right) = \sum_{K \leq L} a_K x^K \tag{4.10}$$

where $K \in \{K_1, \cdots, K_t\}$, and $0 \leq K_j \leq L$, $1 \leq \forall j \leq t$.

Now, the polynomial p consists of t terms: $a_{K_j} x^K = a_{K_j} x_1^{K_j^1} \cdots x_n^{K_j^n}$, where $K_j = \left(K_j^1, \cdots, K_j^n\right)$. Let us denote by $term\left(K_j\right) = a_{K_j} x_1^{K_j^1} \cdots x_n^{K_j^n}$, the jth term of p. Let us denote by $var\left(K_j^i\right) = x_i^{K_j^i}$, $1 \leq i \leq n$, the ith variable in the j term.

We represent all the Bernstein coefficients for $var\left(K_j^i\right)$ as $Imp\left(B_{var\left(K_j^i\right), l_i}\right)$ which is shown as follows:

$$\text{Imp}\left(B_{var\left(K_j^i\right), l_i}\right) = \left[b_{0,l_i}^{var\left(K_j^i\right)}, \cdots, b_{l_i,l_i}^{var\left(K_j^i\right)}\right]. \tag{4.11}$$

We call Imp $\left(B_{var\left(K_j^i\right), l_i}\right)$ in (4.14) the implicit form representation (IBF) of one single variable $var\left(K_j^i\right)$ which is the Bernstein coefficients for that variable. Because the order of this $var\left(K_j^i\right)$ is l_i, we can have up to $l_i + 1$ Bernstein coefficients.

Now, computing all the Bernstein of the j term, $term\left(K_j\right)$, is equal to the cartesian product of the Bernstein coefficients of every single variable Imp $\left(B_{var\left(K_j^i\right), l_i}\right)$ and multiply the resultant multi-dimensional tensor by the coefficient a_{K_j}. However, this process is not memory efficient because the Cartesian product will result into a multi-dimensional tensor which is the drawback of the dense representation. Instead, we compute the IBF of the jth term, $term\left(K_j\right)$, by stacking the IBF of every single variable Imp $\left(B_{var\left(K_j^i\right), l_i}\right)$ row-wise. After that, we multiply the coefficient a_{K_j} by just the first row. All this is summarized in the following equation:

$$\operatorname{Imp}\left(B_{term(K_j),L}\right)=\begin{bmatrix} a_{K_j}\operatorname{Imp}\left(B_{var\left(K_j^1\right),l_1}\right)\\ \vdots \\ \operatorname{Imp}\left(B_{var\left(K_j^n\right),l_n}\right)\end{bmatrix}. \tag{4.12}$$

The length of ith row in $\operatorname{Imp}\left(B_{term(K_j),L}\right)$ is equal to l_i+1, $1\le i\le n$. We denote by $l_{max}=\max\limits_{1\le i\le n} l_i$. The size of the IBF of the jth term, $term\left(K_j\right)$, $\operatorname{Imp}\left(B_{term(K_j),L}\right)$, is equal to $n\times\left(l_{max}+1\right)$, where we pad the rows of lengths $l_i<l_{max}$ with $l_{max}-l_i+1$ zeros at the right side.

Now, the total Bernstein coefficients of the whole polynomial p is the summation of Bernstein coefficients of every term $term\left(K_j\right)$, $1\le j\le t$. This translates to the IBF of the whole polynomial p is by stacking the IBF of every term as follows:

$$\operatorname{Imp}\left(B_{p,L}\right)=\begin{bmatrix}\operatorname{Imp}\left(B_{term(K_1),L}\right)\\ \vdots \\ \operatorname{Imp}\left(B_{term(K_n),L}\right)\end{bmatrix}. \tag{4.13}$$

Now the total size of $\operatorname{Imp}\left(B_{p,L}\right)$ is equal to $nt\times\left(l_{max}+1\right)$. Let see the following example to see how this works.

Example 4.1. *Consider the multivariate polynomial $p(x_1,x_2)=x_1^3x_2^2-30x_1x_2$, defined over the domain $I_2=[1,2]\times[2,4]$, with $L=(3,2)$.*

The IBF of x_1^3, x_2^2, x_1, and x_2 are as follows:

$$\begin{aligned}
Imp\left(B_{var\left(K_j^i\right),l_i}\right) &= \left[1,2,4,8\right] \\
Imp\left(B_{var\left(K_j^i\right),l_i}\right) &= \left[4,8,16\right] \\
Imp\left(B_{var\left(K_j^i\right),l_i}\right) &= \left[-1,-4/3,-5/3,-2\right] \\
Imp\left(B_{var\left(K_j^i\right),l_i}\right) &= \left[2,3,4\right]. \qquad (4.14)
\end{aligned}$$

Using (4.12) *and* (4.13)*, the IBF of the polynomial* p *is written as follows:*

$$Imp(B_{p,L}) = \begin{bmatrix} 1 & 2 & 4 & 8 \\ 4 & 8 & 16 & 0 \\ -30 & -40 & -50 & -60 \\ 2 & 3 & 4 & 0 \end{bmatrix}. \qquad (4.15)$$

This example illustrates that if a polynomial p comprises t terms, and each term is represented by n rows, the total number of rows in the implicit representation amounts to nt. The length of each row is given by $l_{max}+1 = \max\limits_{1\leq i\leq n}(l_i)+1$. Therefore, the overall size of the implicit representation matrix is determined by the dimensions $nt \times (l_{max}+1)$, which leads us to the following proposition:

Proposition 4.2. *Given* n*-dimenstional Bernstein polynomial of a polynomial* p *that comprises of* t *terms and of order* $L = (l_1, \cdots, l_n)$, $B_{p,L}(x)$*. Its implicit representation* $Imp(B_{p,L})$ *is* $2-$*dimensional tensor of a size* $nt \times (l_{max}+1)$*. The total components of the implicit representation is* $O(n)$*.*

Proof. The total components of the implicit representation is $nt \times (l_{max}+1)$ which is $O(n)$.

□

From propositions 4.1 and 4.2, we can notice that the implicit representation reduced the scaling to just be linear with respect to the dimension n compared to the exponential scaling in the dense representation which makes the implicit representation ideal for higher dimensions. In addition the IBF representation is always a 2D tensor no matter what is the dimension or the order of the polynomial compared to its counterpart the dense representation which represents the polynomial as a multi-dimensional tensor.

4.4 Efficient Multiplication of Implicit Bernstein Polynomials

4.4.1 Monomial Bernstein Polynomial Multiplication

In this section, we present a generalized method for the multiplication of implicit Bernstein polynomials in multi-variable settings. This approach extends the techniques for univariate and bivariate cases as described in [35] and [81], respectively. We focus initially on polynomials comprising a single term. Consider two monomials $q_1(x) = a_1 x^{k_1}$ and $q_2(x) = a_2 x^{k_2}$, with $k_1 \leq L_1$ and $k_2 \leq L_2$. The implicit representations of their Bernstein polynomials, $Imp(B_{q_1,L_1})$ and $Imp(B_{q_2,L_2})$, each form a block of size $n \times N$. The implicit representation of the product of these polynomials, $Imp(B_{q_1,L} B_{q_2,L})$, is computed using the following procedure:

$$\mathrm{Imp}\left(\tilde{B}_{q_1,L_1}\right) = \mathrm{Imp}\left(B_{q_1,L_1}\right) * C_{L_1}, \tag{4.16}$$

$$\mathrm{Imp}\left(\tilde{B}_{q_2,L_2}\right) = \mathrm{Imp}\left(B_{q_2,L_2}\right) * C_{L_2}, \tag{4.17}$$

$$\mathrm{Imp}\left(B_{q_1,L_1}B_{q_2,L_2}\right) = \frac{1}{C_{L_1+L_2}} * \mathbf{Conv}\left(\mathrm{Imp}\left(\tilde{B}_{q_1,L_1}\right), \mathrm{Imp}\left(\tilde{B}_{q_2,L_2}\right)\right). \tag{4.18}$$

where C_L denotes the multi-dimensional binomial tensor, with its Kth component in the ith row defined as $(C_L)^i_K = \binom{l_i}{K}$. With some abuse of notation, we use $1/C_{l_i}$ to denote the multi-dimensional binomial tensor where its Kth component in the ith row is equal to $\frac{1}{\binom{l_i}{K}}$. The notation $*$ represents element-wise multiplication, while $\mathbf{Conv}\,(A, B)$ denotes the row-wise convolution between matrices A and B. The above formulation efficiently generalizes the concept of scaled Bernstein polynomials [81] to n-dimensional inputs. The efficient implementation of these operations on GPUs, leveraging element-wise and convolution operations, ensures high computational performance. We denote the process in Equations (4.16)-(4.19) as $\mathbf{Prod\_Bern\_Imp}(B_{p_1,L_1}, B_{p_2,L_2})$.

4.4.2 Multi-variate Bernstein polynomial Multiplication

The method can be extended to handle the multiplication of implicit representations of Bernstein polynomials consisting of multiple terms. Consider two polynomials, $p_1 = \sum_{K \leq L_1} a^1_K x^K$ and $p_2 = \sum_{K \leq L_2} a^2_K x^K$, with t_1 and t_2 terms, respectively. Their implicit representations are denoted as $Imp(B_{p_1,L_1})$ and $Imp(B_{p_2,L_2})$. The multiplication of these polynomials in their implicit Bernstein form is given by:

$$\text{Imp}\left(B_{p_1,L_1}B_{p_2,L_2}\right) = \begin{bmatrix} \vdots \\ \textbf{Prod\_Bern\_Impl}\left(Imp^i\left(B_{p_1,L_1}\right), Imp^j\left(B_{p_2,L_2}\right)\right) \\ \vdots \end{bmatrix},$$

$$1 \leq i \leq t_1, 1 \leq j \leq t_2. \tag{4.19}$$

Here, $Imp^i(B_{p_1,L_1})$ and $Imp^j(B_{p_2,L_2})$ represent the ith and jth sub-matrices (terms) in $Imp(B_{p_1,L_1})$ and $Imp(B_{p_2,L_2})$, respectively. These sub-matrices are obtained by segmenting the original implicit representations into t_1 and t_2 sub-matrices along their rows. This formulation reveals that multiplying multivariate polynomials in implicit Bernstein form effectively boils down to multiplying terms from one polynomial with those from another.

4.5 Efficient Summation of Implicit Bernstein Polynomials

4.5.1 Monomial Bernstein Polynomial Summation

Building upon the foundational work in [35], which addresses the addition of one-dimensional Bernstein polynomials over the unit interval, this section introduces an advanced and generalized approach for the summation of implicit Bernstein polynomials in a multivariate framework. Our extension caters to scenarios with n variables over any specified interval $I_n(\underline{d}, \overline{d})$. Initially, we focus on monomials of the form $q_1(x) = a_1x^{k_1}$ and $q_2(x) = a_2x^{k_2}$, where $k_1 \leq L_1$ and $k_2 \leq L_2$. Their respective implicit Bernstein polynomial representations, $Imp(B_{q_1,L_1})$ and $Imp(B_{q_2,L_2})$, are conceived as $n \times N$ blocks. The summation of these polynomials in their implicit form, $Imp(B_{q_1+q_2,L_{sum}})$, is delineated through the following proposition:

Proposition 4.3. *Given two Bernstein polynomials $B_{q_1,L_1}(x)$ and $B_{q_2,L_2}(x)$ with two different orders $L_1 = (l_1^1, \cdots, l_n^1)$ and $L_2 = (l_1^2, \cdots, l_n^2)$. Define $L_{sum} = \max(L_1, L_2)$, where the* max *operator is applied element-wise. The implicit tensor representation of $B_{q_1+q_2,L_{sum}}$ can be computed as:*

$$L_{sum} = (\max(l_1^1, l_1^2), \ldots, \max(l_n^1, l_n^2)) \tag{4.20}$$

$$Imp\left(B_{q_1,L_{sum}}\right) = \boldsymbol{Prod\_Bern\_Imp}\left(Imp\left(B_{q_1,L_1}\right), 1_{L_{sum}-L_1+1}\right) \tag{4.21}$$

$$Imp\left(B_{q_2,L_{sum}}\right) = \boldsymbol{Prod\_Bern\_Imp}\left(Imp\left(B_{q_2,L_2}\right), 1_{L_{sum}-L_2+1}\right) \tag{4.22}$$

$$Imp\left(B_{q_1+q_2,L_{sum}}\right) = \begin{bmatrix} Imp\left(B_{q_1,L_{sum}}\right) \\ Imp\left(B_{q_2,L_{sum}}\right) \end{bmatrix} \tag{4.23}$$

Here, 1_{L_e-L+1} signifies a two-dimensional tensor with dimensions $n \times (L_e - L + 1)$, exclusively containing ones.

The proof, which extends the argument in [35], is not presented for brevity. In this context, operations (4.21) and (4.22) are recognized as *degree elevation* processes, where tensor dimensions are suitably altered. The final summation, defined by (4.23), is executed by vertically concatenating both tensors once they align dimensionally. We denote this procedure by **Sum_Bern_Imp**.

4.5.2 Multi-variate Bernstein Polynomial Summation

We further extend the method to accommodate summation of implicit Bernstein polynomial representations containing multiple terms. Consider polynomials $p_1 = \sum_{K \leq L_1} a_K^1 x^K$ and $p_2 = \sum_{K \leq L_2} a_K^2 x^K$, with t_1 and t_2 terms, respectively, denoted as $Imp(B_{p_1,L_1})$ and $Imp(B_{p_2,L_2})$. To sum these polynomials in their implicit Bernstein form, we first dissect

$Imp(B_{p_1,L_1})$ and $Imp(B_{p_2,L_2})$ along their primary dimension into t_1 and t_2 sub-matrices, respectively, as $Imp^i(B_{p_1,L_1})$ and $Imp^j(B_{p_2,L_2})$. Subsequent to applying degree elevation via (4.21) and (4.22) to each sub-matrix, we amalgamate the resulting matrices vertically in accordance with (4.23). This efficient and elegant approach encapsulates the core principle of multivariate Bernstein polynomial summation in a multiterm context.

4.6 Optimal under-approximation of ReLU Functions Using Quadratic Polynomials

The BERN-NN model (chapter 3) [36] utilizes a downwards translation of the Bernstein over-approximation of the Rectified Linear Unit (ReLU) σ activation function to achieve its under-approximation. While the over-approximation is optimal (chapter 3) [36]—i.e., produces the tightest over-approximation of the ReLU function—this under-approximation may not always be optimal, especially when the negative side of the pre-input bounds significantly outweighs the positive side. To address this issue, we confine our attention to the use of quadratic polynomials and we formulate the "optimal ReLU under-approximation problem" as an optimization problem defined as follows:

$$\begin{aligned}
&\underset{a,b,c}{\text{minimize}} && A(x) = \int_{\underline{d}}^{\overline{d}} \Big(\sigma(x) - \big(ax^2 + bx + c\big)\Big) dx \\
&\text{subject to} && ax^2 + bx + x \leq \sigma(x), \\
& && x \in [\underline{d}, \overline{d}], \qquad (4.24)
\end{aligned}$$

where $\sigma(x)$ is a ReLU function defined on an interval $[\underline{d}, \overline{d}]$. The optimization problem presented above addresses the problem of finding the coefficients a, b, and c for a quadratic polynomial $q(x) = ax^2 + bx + x$, which provides a tight under-approximation of a given ReLU $\sigma(x)$. The goal is to minimize the area $A(x)$ between the ReLU curve and the under-

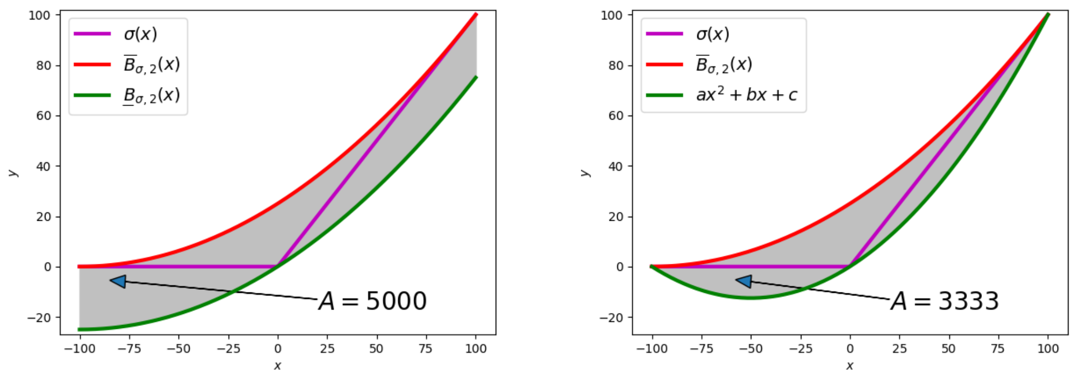


Figure 4.3: (a) State-of-the-art over- and under-approximations of ReLU functions $\sigma(x)$ using high-order polynomials. (b) Proposed optimal over- and under-approximation of ReLU functions $\sigma(x)$. The figure shows the area between the two curves A, indicating the approximation error.

approximation curve over the interval $[\underline{d}, \overline{d}]$, where $\underline{d}$ and $\overline{d}$ are the lower and upper bounds of the ReLU's domain, respectively. By solving this optimization problem, we can obtain a quadratic under-approximation of the ReLU function that accurately captures its behavior over the specified domain. We propose the following algorithm to determine the coefficients for the quadratic polynomial under-approximation of the ReLU function over a given interval. The proofs of this algorithm are detailed in Appendix A. Figure 4.3 vividly illustrates that the approximation error area from the quadratic polynomial is significantly smaller than that of the original under-approximation (chapter 3) [36].

4.7 Ablations

For the ablation studies, we compare the performance and bounds attained by BERN-NN and BERN-NN-IBF. For these comparisons, we attempt to push the approaches to their limits.

The effect of increasing the hidden dimension. Next, we consider a four layer model and keep the input dimension fixed to two and output size one. Each trial varies the dimen-

Algorithm 8 Quadratic Coefficients for ReLU Under-approximation

Input: $I = [\underline{d}, \overline{d}]$
Output: Coefficients a, b, c for the quadratic polynomial under-approximating ReLU

Function: get_quad_coeffs_under($I = [\underline{d}, \overline{d}]$)

```
1: $f_1 \leftarrow 0$
2: $f_2 \leftarrow \frac{2\times\overline{d}^2 - \overline{d}\times\underline{d} - \underline{d}^2}{6.0}$
3: $f_3 \leftarrow \frac{\overline{d}^2 - \underline{d}^2}{2.0}$
4: if $\max(f_1, f_2, f_3) == f_1$ then
5:     return  $a = b = c = 0$
6: else if $\max(f_1, f_2, f_3) == f_2$ then
7:     return  $a = \frac{1}{\overline{d}-\underline{d}}$, $b = \frac{-\underline{d}}{\overline{d}-\underline{d}}$, $c = 0$
8: else
9:     return  $a = 0$, $b = 1$, $c = 0$
10: end if
```

sions of each hidden layer. We see from Figures 4.4 and 4.5 that the performance scaling is similar between the two versions. However, the volume of BERN-NN-IBF is much smaller than BERN-NN. Resulting in a bound that is at least 2x smaller than BERN-NN.

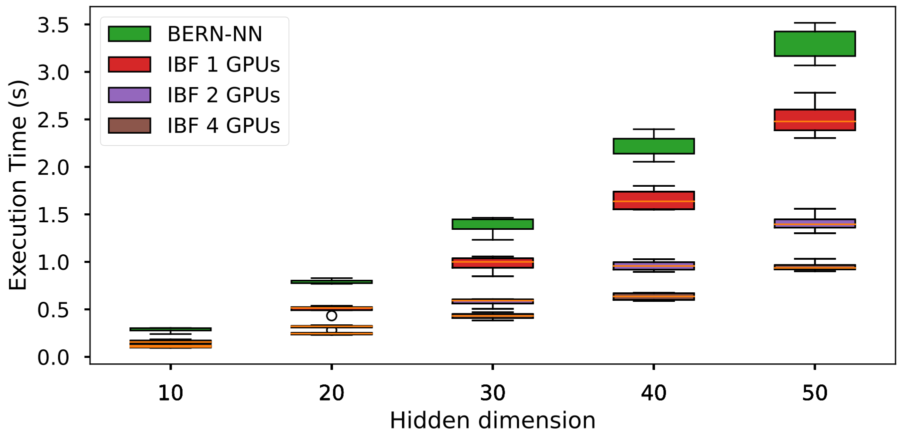


Figure 4.4: Execution time vs. hidden dimension.

The effect of increasing the total number of layers. To study the effect of increasing layers, we keep the input dimension fixed to two with a variable number of layers that each have a hidden size of 5: $[2, 5, \ldots, 5, 1]$. Again, we can see that BERN-NN-IBF achieves better scaling than the original BERN-NN. As we increase the number of layers, we see a

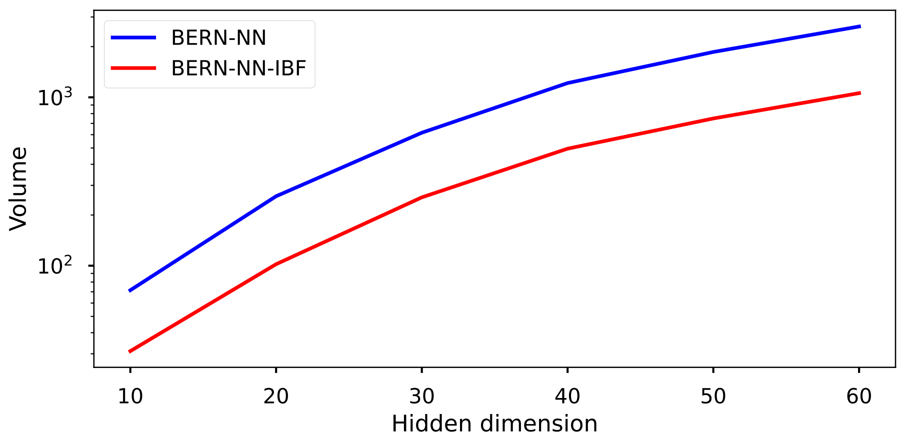


Figure 4.5: Volume vs. hidden dimension.

more obvious performance win for BERN-NN-IBF. Even with this narrow network, we are able to achieve reasonable speedup by distributing over multiple GPUs Results are shown in Figures 4.6 and 4.7

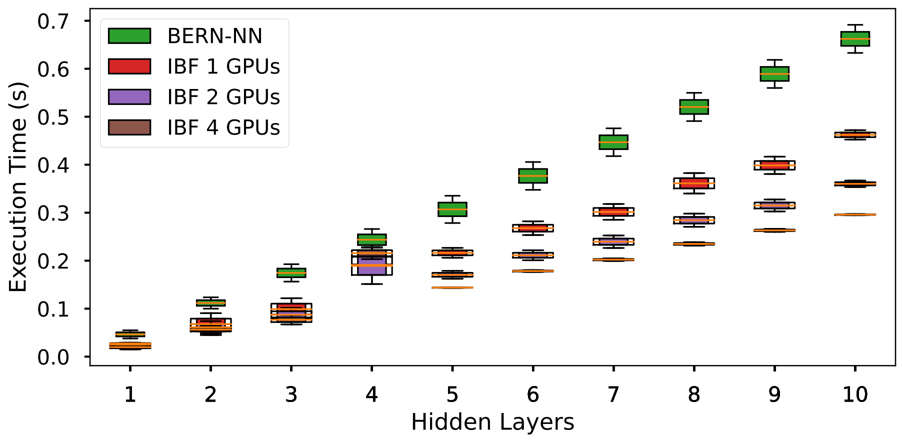


Figure 4.6: Execution time vs. total number of layers.

Increasing the total number of neurons. In figure 4.8, we compute the bounds for progressively larger models to compare the performance of Bern-NN with and without the implicit representation. Again, we observe better scaling with BERN-NN-IBF.

Assuming data is orthant. It is very common that input data can be normalized to fall in the positive quadrant. For instance, in computer vision applications, pixel values may

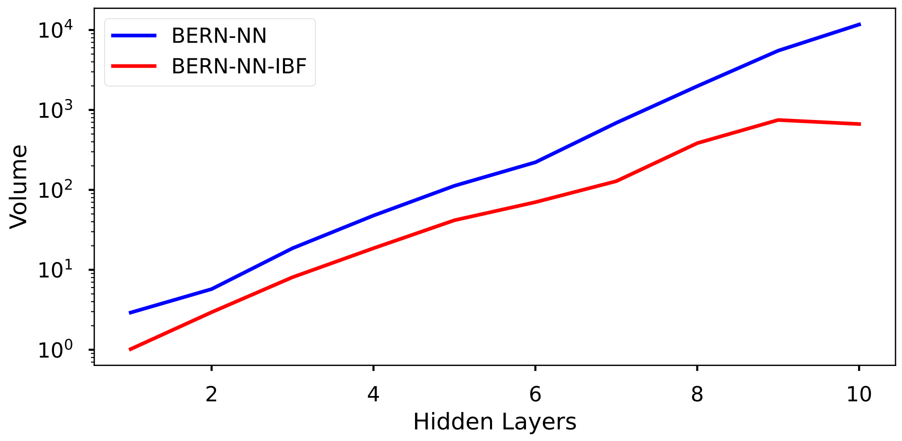


Figure 4.7: Volume vs. total number of layers.

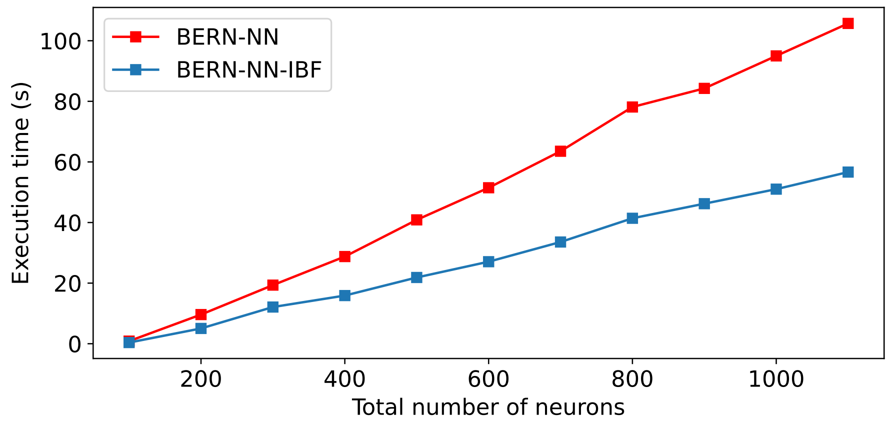


Figure 4.8: Execution time vs. the total number of neurons. Each layer has 100 neurons and we successively add one layer.

be normalized to the range $[0, 1]$. Making the assumption that data falls into the positive quadrant greatly simplifies finding the minimum and maximum of Bernstein polynomials, as we no longer need to convert to the explicit form. Results are shown in Figures 4.9 and 4.10. These experiments use a network with a input dimension of 10 and a hidden layer of varying dimension. We see that making the orthant assumption results in a massive performance boost, with relatively little effect on the resulting bounds in this case.

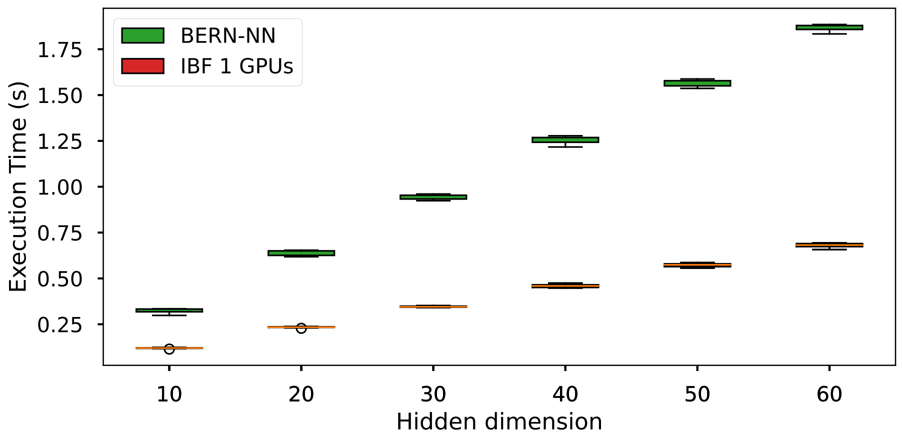


Figure 4.9: Execution time vs. total number of layers.

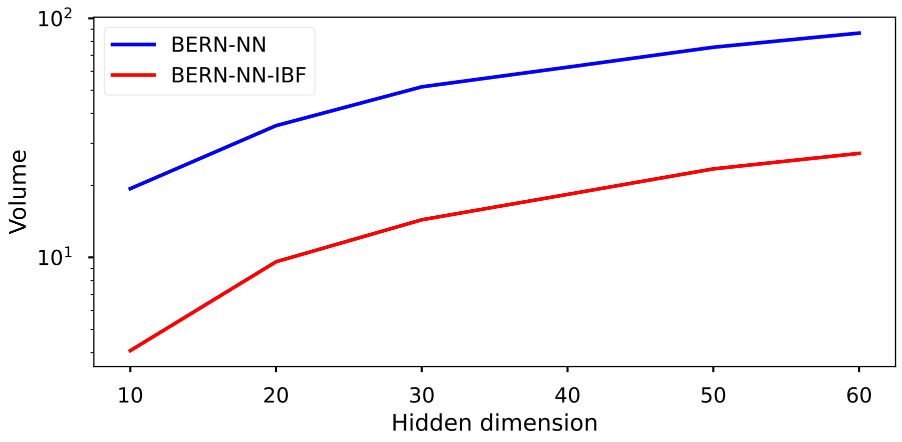


Figure 4.10: Volume vs. total number of layers.

4.8 Tools Comparison

4.8.1 POLAR

In this experiment, we conducted a comprehensive assessment of various tools applied to benchmarks derived from NN controllers, specifically utilized by POLAR, to determine the precision of their estimated bounds. The architecture of the networks employed in each benchmark is detailed in Table 4.1. Furthermore, Table 4.2 encapsulates the performance metrics of these tools, focusing on average execution time and average relative volume across

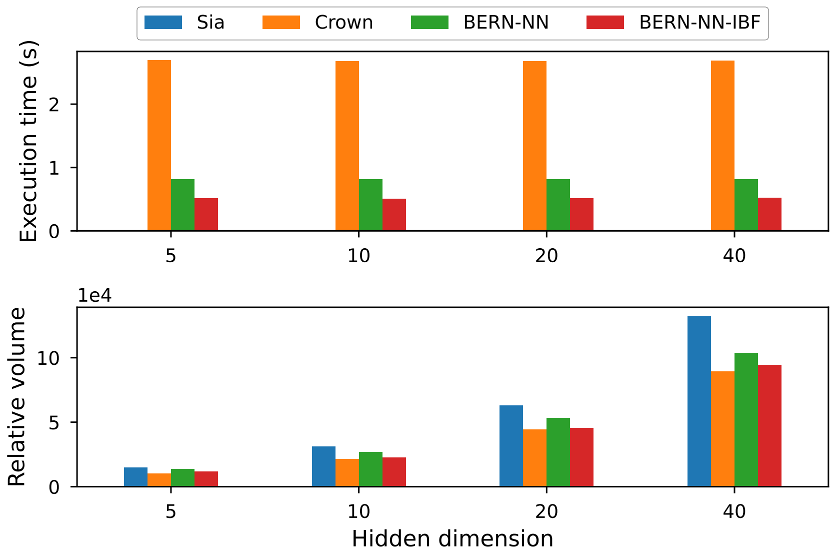


Figure 4.11: We compare the execution time and relative volume as a function of the model's hidden dimension. The time and volume reported are the averages of 10 trials on randomized models. We find that BERN-NN-IBF strikes a good balance between performance and tightness.

six control benchmarks. Notably, BERN-NN-IBF consistently emerged as the second quickest tool, yet it invariably provided the most accurate bounds. This accuracy is particularly significant for control applications, where the specification of interest often spans a time horizon and necessitates multi-step reachability analysis; therefore, achieving finer bounds at each stage is imperative. Moreover, BERN-NN-IBF outperformed CROWN in terms of speed across all benchmarks, with the exception of Benchmark 5. While SIA exhibited faster performance than BERN-NN-IBF, it compromised on the precision of bound estimates. Each benchmark was subjected to tests with five distinct hyperrectangles, all centered at zero, with radii varying within $1, 1.5, 2, 2.5, 3$, to ensure a robust evaluation. This rigorous testing methodology underscores the effectiveness of BERN-NN-IBF in delivering precise and computationally efficient solutions for control applications, highlighting its superiority in optimizing both speed and accuracy in bound estimation.

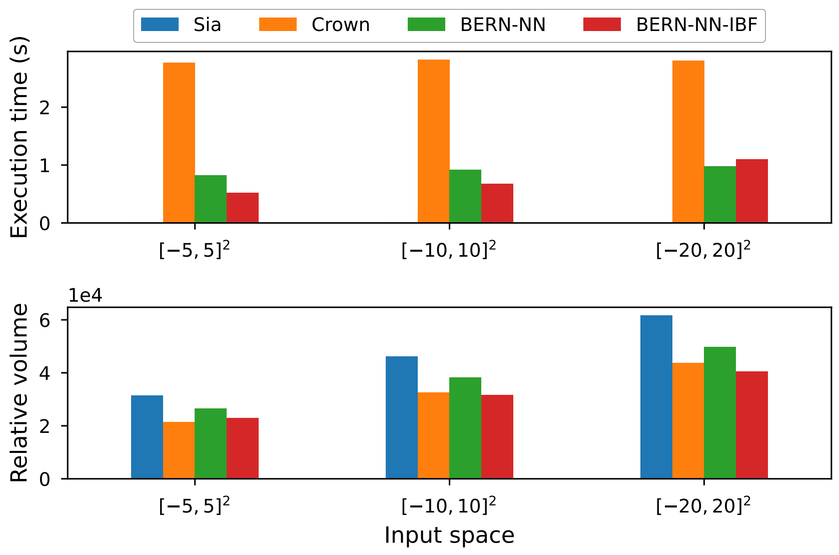


Figure 4.12: We compare the execution time and relative volume as a function of the models inptu dimension. Again, we see that BERN-NN-IBF is fast and able to provide tight bounds.

Table 4.1: POLAR Benchmark Model sizes

	Model Dimensions
Benchmark 1	$[2, 20, 20, 1]$
Benchmark 2	$[2, 20, 20, 1]$
Benchmark 3	$[2, 20, 20, 1]$
Benchmark 4	$[3, 20, 20, 1]$
Benchmark 5	$[4, 100, 100, 1]$
Benchmark 6	$[4, 20, 20, 20, 1]$

Table 4.2: Performance results for execution time and volume for Sia, alpha-CROWN, BERN-NN, and BERN-NN-IBF. These results are the average of five hyper-rectangles with radius $r \in \{1, 1.5, 2, 2.5, 3\}$ for each of the six POLAR benchmarks. Sia is the fastest model, but provides relatively loose bounds. BERN-NN-IBF is always the second fastest model, but consistently provides some of the tightest bounds. All times are reported in seconds.

	Benchmark 1		Benchmark 2		Benchmark 3		Benchmark 4		Benchmark 5		Benchmark 6	
	Time (s)	Vol	Time	Vol	Time	Vol	Time	Vol	Time	Vol	Time	Vol
Sia	**0.013**	2.549	**0.010**	9.790	**0.010**	1.032	**0.010**	8.549	**0.014**	53.391	**0.012**	2.798
Crown	2.389	3.641	2.860	**8.574**	2.875	**0.735**	2.674	15.869	2.912	77.734	3.077	3.264
BERN-NN	0.551	**1.442**	0.846	9.629	0.844	1.106	0.916	8.921	18.367	54.810	1.338	3.454
BERN-NN-IBF	0.218	1.719	0.475	9.003	0.357	0.835	0.304	**7.161**	7.664	**42.747**	0.540	**2.141**

4.8.2 ACAS Xu

In this comprehensive experiment, we meticulously assess the efficacy of BERN-NN-IBF in contrast to CROWN within the context of the unmanned Airborne Collision Avoidance System (ACAS Xu) benchmark, as detailed in [64]. This benchmark encompasses ten distinct properties across 45 neural networks that are instrumental in generating turn advisories for aircraft to avert collisions. Each network comprises 300 neurons distributed over six layers, utilizing ReLU activation functions. The networks are designed with five inputs that represent the states of the aircraft and produce five outputs, with the system adopting the minimum output value as the turn advisory. Further insights into this benchmark are elaborated in the paper [64].

Empirical data presented in Figures 4.13 and 4.14 underscore the superior performance of BERN-NN-IBF over CROWN, both in terms of computational speed and the precision of the volume estimations across all ten specifications. This enhancement in performance is pivotal, particularly in the high-stakes domain of collision avoidance, where the rapid and accurate computation of turn advisories is critical for ensuring the safety of the airspace. BERN-NN-IBF's ability to outperform CROWN in these key areas demonstrates its potential to significantly improve the reliability and efficiency of neural network-based decision-making systems in safety-critical applications.

4.9 GPU Algorithms for Bernstein Polynomial Extrema

4.9.1 Implicit Form Min-Max Computation

The BERN-NN-IBF algorithm involves determining the minimum and maximum values of a Bernstein polynomial stored in implicit form. This can be achieved by computing the

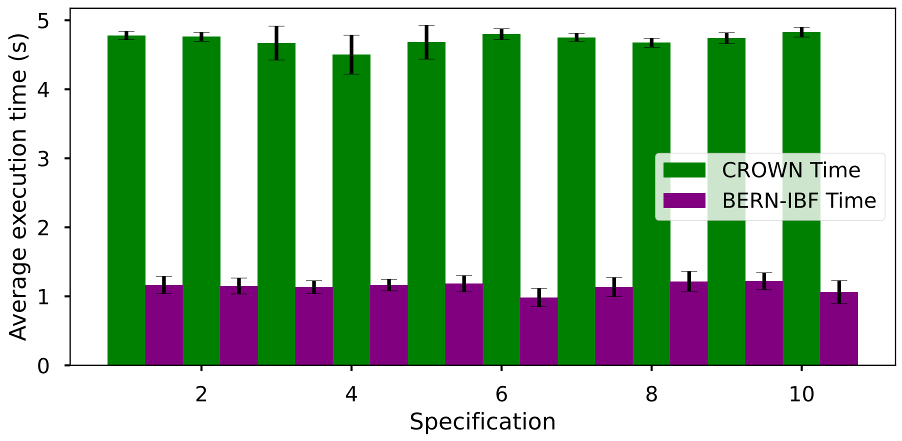


Figure 4.13: Average Execution time of Crown and BERN-NN-IBF on the ACAS Xu benchmark. Error bars represent the standard deviation.

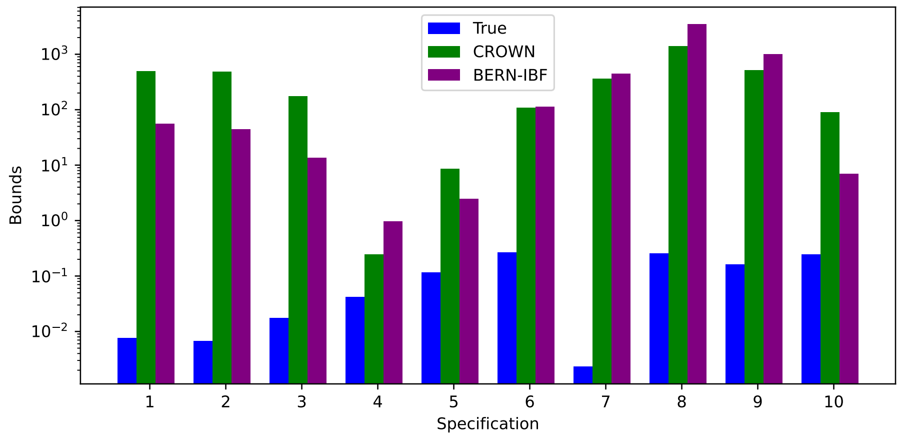


Figure 4.14: Average bound of five neurons across 10 specifications from the ACAS Xu benchmark.

corresponding minimum and maximum coefficients in the polynomials *explicit* form. This conversion can be computationally demanding and memory-intensive, especially for large degrees d and number of variables v. The explicit Bernstein form, denoted by $\text{Den}(B_{q,L})$, is a represented as tensor $T \in \mathbb{R}^{d\times\cdots\times d} = \mathbb{R}^{d^v}$, where the number of axes corresponds to the number of variables. To address potential inefficiencies, we aim to avoid materializing the tensor explicitly.

Algorithm 9 uses the CUDA programming model to efficiently compute the Bernstein polynomial's extrema. Recall that we can index into a tensor $T(i, j, k, \dots)$, which is stored contiguously, by directly accessing memory locations offset by the axis sizes: $T[id^{v-1} + jd^{v-2} + kd^{v-3} + \ldots]$. Each CUDA thread, is associated with an ID, ebf_id in Algorithm 9, that is mapped to a unique set of indices $\{i, j, k, \ldots\}$ to access and compute elements in the explicit Bernstein form. As t, d, and v are known, this mapping is achieved through a set of iterative equations:

$$\begin{aligned} i &= \lfloor t/d^{v-1} \rfloor, \\ j &= \lfloor t/d^{v-2} \rfloor - id, \\ k &= \lfloor t/d^{v-3} \rfloor - id^2 - jd \\ &\ldots \end{aligned} \tag{4.25}$$

The indices are computed on-the-fly within the algorithm, eliminating the need for storage. This iteration corresponds to lines $12 - 14$ of Algorithm 9. Precomputed powers d^{r-1} for $r \in [v]$ are stored in constant global memory and all the threads in a warp compute the same power in each iteration, which ensures low-latency access. Algorithm 9 returns the extrema for the portion of the tensor T that is covered by a CUDA block. Then, we apply an additional reduction to compute the global extrema.

4.9.2 Quadrant-Constrained Min-Max Computation

In models where the polynomials variable are constrained to a single quadrant, specifically when all the variables are positive, we can significantly streamline the computation for both the min and max values. This constraint allows for a simplified kernel, where the computation narrows down to evaluating only two points in the Explicit Bernstein Form (EBF) tensor. This simplification effectively eliminates the need for a loop over the entire EBF tensor,

Algorithm 9 Computing the Extrema of a Bernstein Polynomial in Implicit Form

Input: $T = \text{Imp}(B_{p,L})$, a 3D array representing a Bernstein polynomial. E, the number of elements in the explicit bernstein form
Output: $\min(B_{p,L}), \max(B_{p,L})$ for each CUDA block

Function: ibf-extrema($T = \text{Imp}(B_{p,L})$)

```
1: block_ebf_sum ← zeros($E$, gridDim.y)
2: ebf_id ← global thread id
3: while ebf_id < $E$ do
4:    tsum ← 0
5:    term_id ← blockIdx.y
6:    while term_id < nterms do
7:       accum ← 1
8:       tracker ← 0
9:       for $v \in$ [nvars] do
10:         $p$ ← lookup $d^{\text{nvars}-v-1}$
11:         index ← $\lfloor \text{ebf\_id}/p \rfloor$ − tracker
12:         accum ← accum ∗ $T$[term_id][$v$][index]
13:         tracker ← (tracker + index) ∗ $d$
14:      end for
15:      tsum ← tsum + accum
16:      term_id ← term_id + gridDim.y
17:   end while
18:   block_ebf_sum[ebf_id][blockIdx.y] ← tsum
19:   ebf ← ebf_id + gridDim.x
20: end while
21: return block_ebf_sum
```

reducing the computational complexity to a single iteration over the terms.

This optimization is particularly valuable in scenarios where the positivity of variables can be guaranteed, as is often the case in image data.

4.10 Scaling BERN-NN-IBF Across Multiple GPUs

4.10.1 Distribution Strategy and Challenges

Distributing BERN-NN-IBF across multiple GPUs is essential for handling large models that may not fit within the memory constraints of a single GPU. Our approach involves having each GPU compute bounds for a batch of nodes in a layer, followed by an *allgather* operation to ensure that all GPUs have the input bounds necessary for processing the subsequent layer.

We leverage PyTorch Distributed with the NCCL backend for efficient GPU communication. For the allgather operation, we use `torch.distributed.all_gather_object`. This choice is motivated by the need to communicate Python objects, specifically tensors of different shapes, as part of the bounding process. However, this flexibility comes at a cost of communication inefficiencies because it involves transferring tensors from the GPU to the CPU during the pickling process (i.e., serialization of Python objects into a byte stream which operates on CPU memory). This additional data transfer can be an overhead and impact performance, particularly when working with large polynomials and frequent communication between GPUs. Since `all_gather_object` cannot efficiently communicate objects over NVLink for direct GPU-to-GPU communication, we are likely to encounter a bottleneck that saturates the available communication bandwidth. Thus, our implementation provides an upper bound on strong scaling.

4.10.2 Strong Scaling Experiments

To demonstrate the scalability of our approach, we perform strong scaling experiments on 8 Nvidia A100 GPUs. Results are shown in Figure 4.15.

The batch-parallel computation of node bounds across GPUs accelerates BERN-NN-IBF

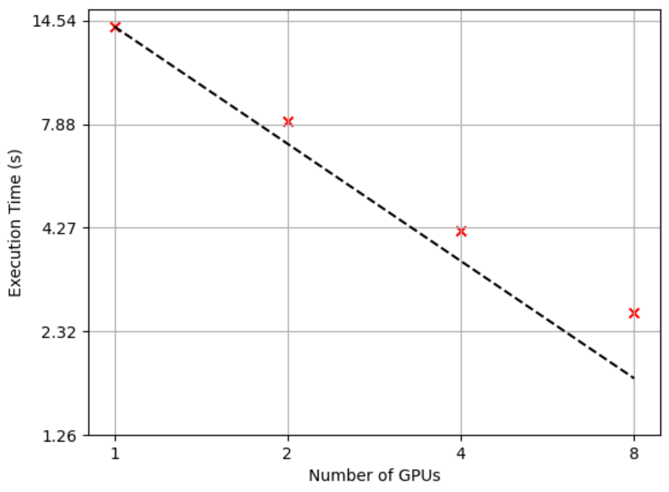


Figure 4.15: Strong Scaling up to eight Nvidia A100 GPUs. We use a fixed model with an input dimension of five, two hidden layers with 100 neurons each, and an output dimension of one. The dashed line represents the ideal strong scaling. The red crosses are the average runtime of 20 trials with the corresponding GPU count. We found that the 95% confidence intervals of the mean runtimes are all within 5% of the mean, so we exclude them from the plot.

while preserving the bounds accuracy. This enables bounding larger models, where intermediate computations do not fit in a single GPU's memory.

Despite the limitations of allgather with pickling, we expect strong scaling due to the computational complexity of computing bounds for each layer. In future work, exploring alternative communication strategies or optimizations tailored for large polynomial data transfers may be essential to further enhance the efficiency and scalability of distributed BERN-NN-IBF.

Bibliography

[1] A. Papachristodoulou, J. Anderson, G. Valmorbida, S. Prajna, P. Seiler, P. A. Parrilo, M. M. Peet and D. Jagt. *SOSTOOLS: Sum of squares optimization toolbox for MATLAB*, 2021.

[2] J. An, N. Zhan, X. Li, M. Zhang, and W. Yi. Model checking bounded continuous-time extended linear duration invariants. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week)*, pages 81–90, 2018.

[3] R. Anderson, J. Huchette, W. Ma, C. Tjandraatmadja, and J. P. Vielma. Strong mixed-integer programming formulations for trained neural networks. *Mathematical Programming*, 183(1):3–39, 2020.

[4] M. Bahavarnia, Y. Shoukry, and N. C. Martins. Controller Synthesis subject to Logical and Structural Constraints: A Satisfiability Modulo Theories (SMT) Approach. In *2020 American Control Conference (ACC)*, pages 5281–5286, 2020.

[5] S. Bak, C. Liu, and T. T. Johnson. The second international verification of neural networks competition (VNN-COMP 2021): Summary and results. *CoRR*, abs/2109.00498:1–15, 2021.

[6] S. Bak, H.-D. Tran, and T. T. Johnson. Numerical verification of affine systems with up to a billion dimensions. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 23–32, 2019.

[7] C. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[8] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi. Measuring neural net robustness with constraints. In *Advances in Neural Information Processing Systems*, volume 29, pages 2613–2621, Barcelona, Spain, 2016. Association for Computing Machinery.

[9] A. Bauer, M. Pister, and M. Tautschnig. Tool-support for the analysis of hybrid systems and models. In *2007 Design, Automation Test in Europe Conference Exhibition*, pages 1–6, 2007.

[10] M. Behroozi. Largest Inscribed Rectangles in Geometric Convex Sets. *CoRR*, abs/1905.13246, 2019.

[11] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[12] M. A. Ben Sassi, R. Testylier, T. Dang, and A. Girard. Reachability analysis of polynomial systems using linear programming relaxations. In *International Symposium on Automated Technology for Verification and Analysis*, pages 137–151. Springer, 2012.

[13] M. Boreale. Algorithms for exact and approximate linear abstractions of polynomial continuous systems. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week)*, pages 207–216, 2018.

[14] E. Botoeva, P. Kouvaros, J. Kronqvist, A. Lomuscio, and R. Misener. Efficient verification of relu-based neural networks via dependency analysis. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(4):3291–3299, 2020.

[15] N. Brisebarre, M. Joldeş, É. Martin-Dorel, M. Mayero, J.-M. Muller, I. Paşca, L. Rideau, and L. Théry. Rigorous polynomial approximation using Taylor models in Coq. In *NASA Formal Methods Symposium*, pages 85–99. Springer, 2012.

[16] C. W. Brown. Improved projection for cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 32(5):447–465, 2001.

[17] R. Bunel, J. Lu, I. Turkaslan, P. Kohli, P. Torr, and P. Mudigonda. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research*, 21(42):1–39, 2020.

[18] X. Chen. *Reachability analysis of non-linear hybrid systems using taylor models.* PhD thesis, Fachgruppe Informatik, RWTH Aachen University, 2015.

[19] X. Chen, E. Ábrahám, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *International Conference on Computer Aided Verification*, pages 258–263. Springer, 2013.

[20] C.-H. Cheng, G. Nührenberg, and H. Ruess. Maximum resilience of artificial neural networks. In D. D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis*, pages 251–268, N/A, 2017. Springer, N/A.

[21] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Automata Theory and Formal Languages 2nd GI Conference Kaisers lautern, May 20–23, 1975*, pages 134–183. Springer, 1975.

[22] L. De Branges. The stone-weierstrass theorem. *Proceedings of the American Mathematical Society*, 10(5):822–824, 1959.

[23] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[24] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

[25] T. Dreossi. Sapo: Reachability computation and parameter synthesis of polynomial dynamical systems. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*, pages 29–34, 2017.

[26] B. Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014.

[27] S. Dutta, X. Chen, S. Jha, S. Sankaranarayanan, and A. Tiwari. Sherlock-a tool for verification of neural network feedback systems: demo abstract. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 262–263, N/A, 2019. N/A.

[28] S. Dutta, X. Chen, and S. Sankaranarayanan. Reachability analysis for neural feedback systems using regressive polynomial rule inference. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 157–168, 2019.

[29] S. Dutta, X. Chen, and S. Sankaranarayanan. Reachability analysis for neural feedback systems using regressive polynomial rule inference. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 157–168, N/A, 2019. N/A.

[30] K. Dvijotham, R. Stanforth, S. Gowal, T. A. Mann, and P. Kohli. A dual approach to scalable verification of deep networks. In A. Globerson and R. Silva, editors, *Uncertainty in Artificial Intelligence*, volume 1, pages 550–559, N/A, 2018. N/A.

[31] M. England and J. H. Davenport. The complexity of cylindrical algebraic decomposition with respect to polynomial degree. In *International Workshop on Computer Algebra in Scientific Computing*, pages 172–192. Springer, 2016.

[32] J. Fan, C. Huang, X. Chen, W. Li, and Q. Zhu. Reachnn*: A tool for reachability analysis of neural-network controlled systems. In *International Symposium on Automated Technology for Verification and Analysis*, pages 537–542, N/A, 2020. Springer, N/A.

[33] R. T. Farouki. The bernstein polynomial basis: A centennial retrospective. *Computer Aided Geometric Design*, 29(6):379–419, 2012.

[34] R. T. Farouki and V. Rajan. On the numerical condition of polynomials in Bernstein form. *Computer Aided Geometric Design*, 4(3):191–216, 1987.

[35] R. T. Farouki and V. Rajan. Algorithms for polynomials in Bernstein form. *Computer Aided Geometric Design*, 5(1):1–26, 1988.

[36] W. Fatnassi, H. Khedr, V. Yamamoto, and Y. Shoukry. Bern-nn: Tight bound propagation for neural networks using bernstein polynomial interval arithmetic. In *Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control*, pages 1–11, 2023.

[37] M. Fazlyab, A. Robey, H. Hassani, M. Morari, and G. Pappas. Efficient and accurate estimation of lipschitz constants for deep neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, pages 11423–11434, N/A, 2019. Curran Associates, Inc.

[38] J. Ferlez and Y. Shoukry. AReN: assured ReLU NN architecture for model predictive control of LTI systems. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, pages 1–11, 2020.

[39] H. J. Ferreau, S. Almér, H. Peyrl, J. L. Jerez, and A. Domahidi. Survey of industrial applications of embedded model predictive control. In *2016 European Control Conference (ECC)*, pages 601–601. IEEE, 2016.

[40] M. Fischetti and J. Jo. Deep neural networks and mixed integer linear optimization. *Constraints*, 23(3):296–309, 2018.

[41] R. FitzHugh. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical journal*, 1(6):445–466, 1961.

[42] I. A. Fotiou, P. Rostalski, P. A. Parrilo, and M. Morari. Parametric optimization and optimal control using algebraic geometry methods. *International Journal of Control*, 79(11):1340–1358, 2006.

[43] D. J. Fremont, J. Chiu, D. D. Margineantu, D. Osipychev, and S. A. Seshia. Formal analysis and redesign of a neural network-based aircraft taxiing system with verifai. In *International Conference on Computer Aided Verification*, pages 122–134, N/A, 2020. Springer, N/A.

[44] J. Garloff. Convergent bounds for the range of multivariate polynomials. In *International Symposium on Interval Mathematics*, pages 37–56. Springer, 1985.

[45] J. Garloff. Convergent bounds for the range of multivariate polynomials. In *International Symposium on Interval Mathematics*, pages 37–56. Springer, 1985.

[46] J. Garloff and A. P. Smith. An improved method for the computation of affine lower bound functions for polynomials. In *Frontiers in Global Optimization*, pages 135–144. Springer, 2004.

[47] J. Garloff and A. P. Smith. Guaranteed affine lower bound functions for multivariate polynomials. In *PAMM: Proceedings in Applied Mathematics and Mechanics*, volume 7, pages 1022905–1022906, N/A, 2007. Wiley Online Library, N/A.

[48] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. Exact Combinatorial Optimization with Graph Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 32*, 2019.

[49] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18, N/A, 2018. IEEE, N/A.

[50] D. Goldfarb and A. Idnani. A numerically stable dual method for solving strictly convex quadratic programs. *Mathematical programming*, 27(1):1–33, 1983.

[51] P. Henriksen and A. Lomuscio. Deepsplit: An efficient splitting method for neural network verification via indirect effect analysis. In *IJCAI*, pages 2549–2555, N/A, 2021. N/A.

[52] H. Hong. An Improvement of the Projection Operator in Cylindrical Algebraic Decomposition. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, ISSAC '90, page 261–264, New York, NY, USA, 1990. Association for Computing Machinery.

[53] C. Huang, J. Fan, X. Chen, W. Li, and Q. Zhu. Polar: A polynomial arithmetic framework for verifying neural-network controlled systems. In *International Symposium on Automated Technology for Verification and Analysis*, pages 414–430, N/A, 2022. Springer, N/A.

[54] G. Irving, C. Szegedy, A. A. Alemi, N. Eén, F. Chollet, and J. Urban. Deepmath-deep sequence models for premise selection. *Advances in Neural Information Processing Systems*, 29:2235–2243, 2016.

[55] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee. Verisig: verifying safety properties of hybrid systems with neural network controllers. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '19, pages 169–178, New York, NY, USA, 2019. Association for Computing Machinery.

[56] C. Kaliszyk, F. Chollet, and C. Szegedy. Holstep: A machine learning dataset for higher-order logic theorem proving. *arXiv preprint arXiv:1703.00426*, 2017.

[57] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In R. Majumdar and V. Kunčak, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 97–117, N/A, 2017. Springer International Publishing.

[58] H. K. Khalil. *Nonlinear systems; 3rd ed.* Prentice-Hall, Upper Saddle River, NJ, 2002.

[59] H. Khedr, J. Ferlez, and Y. Shoukry. Peregrinn: Penalized-relaxation greedy neural network verifier. In A. Silva and K. R. M. Leino, editors, *Computer Aided Verification*, pages 287–300, Cham, 2021. Springer International Publishing.

[60] N. Kochdumper and M. Althoff. Reachability analysis for hybrid systems with nonlinear guard sets. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, pages 1–10, 2020.

[61] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[62] L. Lestingi, M. Askarpour, M. M. Bersani, and M. Rossi. Formal verification of human-robot interaction in healthcare scenarios. In *International Conference on Software Engineering and Formal Methods*, pages 303–324. Springer, 2020.

[63] C. Liu, T. Arnon, C. Lazarus, C. Strong, C. Barrett, M. J. Kochenderfer, et al. Algorithms for verifying deep neural networks. *Foundations and Trends® in Optimization*, 4(3-4):244–404, 2021.

[64] D. M. Lopez, T. T. Johnson, S. Bak, H.-D. Tran, and K. Hobbs. Evaluation of neural network verification methods for air to air collision avoidance. *AIAA Journal of Air Transportation (JAT)*, 2022.

[65] A. Mahboubi. Programming and certifying a CAD algorithm in the Coq system. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[66] C. U. Manual. Ibm ilog cplex optimization studio. *Version*, 12:1987–2018, 1987.

[67] T. Marcucci and R. Tedrake. Mixed-integer formulations for optimal control of piecewise-affine systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 230–239, 2019.

[68] S. McCallum. An improved projection operation for cylindrical algebraic decomposition. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 242–268. Springer, 1998.

[69] B. Mourrain and J. P. Pavone. Subdivision methods for solving polynomial equations. *Journal of Symbolic Computation*, 44(3):292–306, 2009.

[70] C. Munoz and A. Narkawicz. Formalization of Bernstein polynomials and applications to global optimization. *Journal of Automated Reasoning*, 51(2):151–196, 2013.

[71] C. A. Munoz. Formal Methods in Air Traffic Management: The Case of Unmanned Aircraft Systems. In *International Colloquium on Theoretical Aspects of Computing (ICTAC 2015)*, 2015.

[72] A. Narkawicz and C. A. Munoz. Formal Verification of Conflict Detection Algorithms for Arbitrary Trajectories. *Reliab. Comput.*, 17:209–237, 2012.

[73] S. D. Nelson and C. Pecheur. Formal verification for a next-generation space shuttle. In *International Workshop on Formal Approaches to Agent-Based Systems*, pages 53–67. Springer, 2002.

[74] D. A. Popescu and R. T. Garcia. Multivariate polynomial multiplication on GPU. *Procedia Computer Science*, 80:154–165, 2016.

[75] M. Rabi. Piece-wise analytic trajectory computation for polytopic switching between stable affine systems. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, pages 1–11, 2020.

[76] A. V. Rao. A survey of numerical methods for optimal control. *Advances in the Astronautical Sciences*, 135(1):497–528, 2009.

[77] S. Ray and P. Nataraj. A Matrix Method for Efficient Computation of Bernstein Coefficients. *Reliab. Comput.*, 17(1):40–71, 2012.

[78] S. Ray and P. Nataraj. A Matrix Method for Efficient Computation of Bernstein Coefficients. *Reliab. Comput.*, 17(1):40–71, 2012.

[79] D. Roy, M. Balszun, T. Heurung, and S. Chakraborty. Multi-Domain Coupling for Automated Synthesis of Distributed Cyber-Physical Systems. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2018.

[80] S. Sadraddini and R. Tedrake. Robust output feedback control with guaranteed constraint satisfaction. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, pages 1–10, 2020.

[81] J. Sánchez-Reyes. Algebraic manipulation in the Bernstein form made simple via convolutions. *Computer-Aided Design*, 35(10):959–967, 2003.

[82] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill. Learning a SAT Solver from Single-Bit Supervision. In *International Conference on Learning Representations*, 2019.

[83] Z. Shen, H. Yang, and S. Zhang. Deep network approximation characterized by number of neurons. *arXiv preprint arXiv:1906.05497*, 2019.

[84] Y. Shoukry, M. Chong, M. Wakaiki, P. Nuzzo, A. Sangiovanni-Vincentelli, S. A. Seshia, J. P. Hespanha, and P. Tabuada. SMT-based observer design for cyber-physical systems under sensor attacks. *ACM Transactions on Cyber-Physical Systems*, 2(1):1–27, 2018.

[85] A. P. Smith. Fast construction of constant bound functions for sparse polynomials. *Journal of Global Optimization*, 43(2):445–458, 2009.

[86] A. P. Smith. Fast construction of constant bound functions for sparse polynomials. *Journal of Global Optimization*, 43(2):445–458, 2009.

[87] V. Stahl. *Interval methods for bounding the range of polynomials and solving systems of nonlinear equations*. PhD thesis, Johannes Kepler University Linz, 1995.

[88] X. Sun, H. Khedr, and Y. Shoukry. Formal verification of neural network controlled autonomous systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 147–156, 2019.

[89] X. Sun, H. Khedr, and Y. Shoukry. Formal verification of neural network controlled autonomous systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 147–156, N/A, 2019. N/A.

[90] X. Sun and Y. Shoukry. Neurosymbolic motion and task planning for linear temporal logic tasks. *arXiv preprint arXiv:2210.05180*, N/A(N/A):N/A, 2022.

[91] V. Tjeng, K. Y. Xiao, and R. Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*, number N/A, page N/A, N/A, 2019. N/A.

[92] H.-D. Tran, X. Yang, D. Manzanas Lopez, P. Musau, L. V. Nguyen, W. Xiang, S. Bak, and T. T. Johnson. Nnv: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification*, pages 3–17, N/A, 2020. Springer International Publishing.

[93] P. Trébuchet, B. Mourrain, and M. A. Bucero. Border Basis for Polynomial System Solving and Optimization. pages 212–220, 2016.

[94] L. Vandenberghe. The CVXOPT linear and quadratic cone program solvers. *Online: http://cvxopt. org/documentation/coneprog. pdf*, 2010.

[95] A. P. Vinod and M. M. Oishi. Scalable underapproximative verification of stochastic LTI systems using convexity and compactness. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week)*, pages 1–10, 2018.

[96] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Efficient formal safety analysis of neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31, pages 6367–6377, N/A, 2018. N/A.

[97] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal security analysis of neural networks using symbolic intervals. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 1599–1614, N/A, 2018. USENIX Association.

[98] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and J. Z. Kolter. Beta-CROWN: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. In *Advances in Neural Information Processing Systems*, number N/A, page N/A, N/A, 2021. N/A.

[99] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and J. Z. Kolter. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. *Advances in Neural Information Processing Systems*, 34:29909–29921, 2021.

[100] E. Wong and Z. Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International conference on machine learning*, pages 5286–5295, N/A, 2018. N/A.

[101] W. Xiang, H.-D. Tran, and T. T. Johnson. Output reachable set estimation and verification for multilayer neural networks. *IEEE transactions on neural networks and learning systems*, 29(11):5777–5783, 2018.

[102] W. Xiang, H.-D. Tran, J. A. Rosenfeld, and T. T. Johnson. Reachable set estimation and safety verification for piecewise linear systems with neural network controllers. In *2018 Annual American Control Conference (ACC)*, pages 1574–1579, N/A, 2018. N/A.

[103] K. Xu, H. Zhang, S. Wang, Y. Wang, S. Jana, X. Lin, and C.-J. Hsieh. Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. In *ICLR*, number N/A, page N/A, N/A, 2021. N/A.

[104] B. Xue, M. Fränzle, and N. Zhan. Under-approximating reach sets for polynomial continuous systems. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week)*, pages 51–60, 2018.

[105] B. Xue, Q. Wang, N. Zhan, and M. Fränzle. Robust invariant sets generation for state-constrained perturbed polynomial systems. In *Proceedings of the 22nd ACM international conference on hybrid systems: Computation and control*, pages 128–137, 2019.

[106] W. F. Yasser Shoukry. PolyAR: A Highly Parallelizable Solver For Polynomial Inequality Constraints Using Convex Abstraction Refinement. In *IFAC-PapersOnLine*, volume 54, pages 43–48, 2021.

[107] M. Zettler and J. Garloff. Robustness analysis of polynomials with polynomial parameter dependency using Bernstein expansion. *IEEE Transactions on Automatic Control*, 43(3):425–431, 1998.

Appendix A

Proof of Optimal Under-approximation of ReLU

To ensure that the quadratic polynomial under-approximates the ReLU function over the entire interval $[\underline{d}, \overline{d}]$, the optimization problem in (4.24) includes the constraint $q(x) = ax^2 + bx + c \leq \sigma(x)$, $\forall x \in [\underline{d}, \overline{d}]$. This constraint guarantees that the under-approximation curve always lies below the ReLU curve. This quadratic constraint depends on the variable x, making it harder to solve the optimization problem. In the following theorem, we linearize this constraint and remove this dependency so that it depends only on the polynomial coefficients a, b, and c:

Proposition A.1. *The following conditions are sufficient to ensure that the quadratic polynomial $q\left(x\right) = ax^2 + bx + c$ is an under approximation of the ReLU function—i.e., $q(x) \leq$*

$\sigma(x)$:

$$\begin{aligned}
q\left(\underline{d}\right) = a\underline{d}^2 + b\underline{d} + c &\leq 0 \\
q\left(0\right) = c &\leq 0 \\
q\left(\overline{d}\right) = a\overline{d}^2 + b\overline{d} + c &\leq \overline{d} \\
0 &\leq a
\end{aligned} \tag{A.1}$$

Proof. Our goal is to show that $q(x) \leq \sigma(x)$ for all $x \in [\underline{d}, \overline{d}]$, subject to the given conditions. We divide the interval $[\underline{d}, \overline{d}]$ into two cases:

Case 1: For $x \in [\underline{d}, 0]$, we have $ax^2 \leq a\underline{d}^2$. If $b \geq 0$, then $q(x) = ax^2 + bx + c \leq q(\underline{d}) = a\underline{d}^2 + c \leq a\underline{d}^2 + b\underline{d} + c \leq 0$. If $b \leq 0$, then $q(x) = ax^2 + bx + c \leq q(\underline{d}) = a\underline{d}^2 + b\underline{d} + c \leq 0$. In either case, we have $q(x) \leq \sigma(x)$ for all $x \in [\underline{d}, 0]$.

Case 2: For $x \in [0, \overline{d}]$, we define $l(x) = m_1 x + m_2$, where $m_1 = \frac{q(\overline{d}) - q(0)}{\overline{d}}$ and $m_2 = q(0)$. It is clear that $l(0) = q(0)$ and $l(\overline{d}) = q(\overline{d})$. Furthermore, the quadratic polynomial q is convex because $a \geq 0$. Therefore, $q(x) \leq l(x)$ for all $x \in [0, \overline{d}]$. To complete the proof, we must show that $l(x) \leq x$ for all $x \in [0, \overline{d}]$.

We define $l_{diff}(x) = l(x) - x = \frac{q(\overline{d}) - q(0) - \overline{d}}{\overline{d}} x + q(0)$. Then, we have $l_{diff}(0) = q(0) \leq 0$ and $l_{diff}(\overline{d}) = q(\overline{d}) - \overline{d} \leq 0$. Since l_{diff} is a line defined over $[0, \overline{d}]$ and $l_{diff}(0) \leq 0$ and $l_{diff}(\overline{d}) \leq 0$, we conclude that $l_{diff}(x) \leq 0$ for all $x \in [0, \overline{d}]$. Therefore, $l(x) \leq x$ for all $x \in [0, \overline{d}]$. Consequently, we have $q(x) \leq \sigma(x)$ for all $x \in [0, \overline{d}]$.

Thus, we have shown that $q(x) \leq \sigma(x)$ for all $x \in [\underline{d}, \overline{d}]$, as required. □

Proposition A.1 enables us to transform the optimization problem's non-linear constraint into four linear constraints. We reformulate the optimization problem (4.24) as:

$$\begin{aligned}
\underset{a,b,c}{\text{minimize}} \quad & A(x) = \int_{\underline{d}}^{\overline{d}} \Big(\sigma(x) - \big(ax^2 + bx + c\big)\Big) dx \\
\text{subject to} \quad & a\underline{d}^2 + b\underline{d} + c \leq 0, \\
& a\overline{d}^2 + b\overline{d} + c \leq \overline{d}, \\
& c \leq 0, \\
& 0 \leq a.
\end{aligned} \tag{A.2}$$

The objective function $A(x)$ of the optimization problem can be expressed in the following form:

$$\begin{aligned}
A(x) &= \int_{\underline{d}}^{\overline{d}} \Big(\sigma(x) - \big(ax^2 + bx + c\big)\Big) \\
&= \int_{\underline{d}}^{\overline{d}} \sigma(x)\, dx - \int_{\underline{d}}^{\overline{d}} \big(ax^2 + bx + c\big)\, dx \\
&= \frac{\overline{d}}{2} - \left(a\left(\frac{\overline{d}^3 - \underline{d}^3}{3}\right) + b\left(\frac{\overline{d}^2 - \underline{d}^2}{2}\right) + c\left(\overline{d} - \underline{d}\right)\right).
\end{aligned} \tag{A.3}$$

By examining eq. (A.3), it becomes apparent that the objective function is linear with respect to the coefficients a, b, and c. Hence, we can transform the optimization problem in (4.24) into a linear programming (LP) problem:

$$\begin{aligned}
\underset{a,b,c}{\text{maximize}} \quad & a\left(\frac{\overline{d}^3 - \underline{d}^3}{3}\right) + b\left(\frac{\overline{d}^2 - \underline{d}^2}{2}\right) + c\left(\overline{d} - \underline{d}\right) \\
\text{subject to} \quad & a\underline{d}^2 + b\underline{d} + c \leq 0, \\
& a\overline{d}^2 + b\overline{d} + c \leq \overline{d}, \\
& c \leq 0, \\
& 0 \leq a.
\end{aligned} \tag{A.4}$$

We now focus on optimizing the problem further. A critical observation based on the inequalities $\frac{\overline{d}^3-\underline{d}^3}{3} \geq 0$, $\frac{\overline{d}^2-\underline{d}^2}{2} \geq 0$, $\overline{d}-\underline{d} \geq 0$, and $c \leq 0$, allows us to consider the following inequality:

$$\begin{aligned}
&\forall a, b, \text{ and } c \in \mathbb{R}: \\
&a\left(\frac{\overline{d}^3-\underline{d}^3}{3}\right)+b\left(\frac{\overline{d}^2-\underline{d}^2}{2}\right)+c(\overline{d}-\underline{d}) \\
&\leq a\left(\frac{\overline{d}^3-\underline{d}^3}{3}\right)+b\left(\frac{\overline{d}^2-\underline{d}^2}{2}\right).
\end{aligned}$$

The right side of the inequality is attained when $c=0$. This adjustment simplifies our linear program (LP) by effectively reducing the number of variables, thereby transforming it into a more manageable form. The revised LP, with $c=0$, is:

$$\begin{aligned}
\underset{a,b}{\text{maximize}} \quad & a\left(\frac{\overline{d}^3-\underline{d}^3}{3}\right)+b\left(\frac{\overline{d}^2-\underline{d}^2}{2}\right) \\
\text{subject to} \quad & a\underline{d}^2+b\underline{d} \leq 0, \\
& a\overline{d}^2+b\overline{d} \leq \overline{d}, \\
& 0 \leq a. \qquad (A.5)
\end{aligned}$$

This step is crucial as it reduces the LP's dimensionality from three to two, making the problem less complex and more approachable. The number of constraints also drops from four to three, further simplifying the solution process. These reductions are strategic, streamlining the problem without sacrificing the integrity of the solution space.

We concentrate on solving the LP problem delineated in (A.5). For simplification, let's define the function $f(a,b)$ representing our objective:

$$f(a,b)=a\left(\frac{\overline{d}^3-\underline{d}^3}{3}\right)+b\left(\frac{\overline{d}^2-\underline{d}^2}{2}\right)$$

The problem's feasible region is notably a triangle, defined by three vertices: $v_1 = (0,0)$, $v_2 = \left(\frac{1}{\bar{d}-\underline{d}}, \frac{-\underline{d}}{\bar{d}-\underline{d}}\right)$, and $v_3 = (0,1)$. Our goal is to identify the maximum value of $f(a,b)$ at these specific points, which correspond to potential solutions of the LP.

We proceed by evaluating f at each vertex and comparing these values. The optimal solution is determined based on which vertex yields the maximum value. Here are the conditions:

let us define: $f_1 = f(v_1)$, $f_2 = f(v_2)$, and $f_3 = f(v_3)$. If f_1 is the maximum among f_1, f_2, and f_3, then the optimal values are $a = 0$, $b = 0$, and $c = 0$. If f_2 is the maximum, then the optimal solution becomes $a = \frac{1}{\bar{d}-\underline{d}}$, $b = \frac{-\underline{d}}{\bar{d}-\underline{d}}$, and $c = 0$. If $f(v_3)$ is the maximum, we set $a = 0$, $b = 1$, and $c = 0$ for the optimal solution. You can find more details in Algorithm 1.

This approach simplifies the solution process by reducing the problem to comparisons of function values at specific points rather than requiring a more extensive search through a continuous space. The solutions adapt based on the vertex yielding the highest function value, guiding the parameters a, b, and c accordingly.