Reachability Analysis for Neural Feedback Systems Using Regressive Polynomial Rule Inference

Souradeep Dutta

Xin Chen

Sriram Sankaranarayanan

Reachability Analysis for Neural Feedback Systems using Regressive Polynomial Rule Inference

Souradeep Dutta
souradeep.dutta@colorado.edu
University of Colorado, Boulder
Boulder, Colorado

Xin Chen
xchen4@udayton.edu
University of Dayton
Dayton, Ohio

Sriram Sankaranarayanan
srirams@colorado.edu
University of Colorado, Boulder
Boulder, Colorado

ABSTRACT

We present an approach to construct reachable set overapproximations for continuous-time dynamical systems controlled using neural network feedback systems. Feedforward deep neural networks are now widely used as a means for learning control laws through techniques such as reinforcement learning and data-driven predictive control. However, the learning algorithms for these networks do not guarantee correctness properties on the resulting closed-loop systems. Our approach seeks to construct overapproximate reachable sets by integrating a Taylor model-based flowpipe construction scheme for continuous differential equations with an approach that replaces the neural network feedback law for a small subset of inputs by a polynomial mapping. We generate the polynomial mapping using regression from input-output samples. To ensure soundness, we rigorously quantify the gap between the output of the network and that of the polynomial model. We demonstrate the effectiveness of our approach over a suite of benchmark examples ranging from 2 to 17 state variables, comparing our approach with alternative ideas based on range analysis.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Embedded and cyber-physical systems**; • **Mathematics of computing** → *Interval arithmetic*; *Differential equations*;

KEYWORDS

reachability analysis, polynomial regression , neural network, hybrid system, flowpipe construction

ACM Reference Format:
Souradeep Dutta, Xin Chen, and Sriram Sankaranarayanan. 2019. Reachability Analysis for Neural Feedback Systems using Regressive Polynomial Rule Inference. In *22nd ACM International Conference on Hybrid Systems: Computation and Control (HSCC '19), April 16–18, 2019, Montreal, QC, Canada.* ACM, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.1145/3302504.3313351

1 INTRODUCTION

We present a reachability analysis approach for neural feedback systems consisting of nonlinear ODEs with deep neural networks as feedback. Given initial conditions and a range of possible time-varying disturbances, our approach computes an overapproximation of the reachable sets over a finite time horizon. The overapproximation can be used to prove safety properties of the system by excluding a given target set. Additionally, reachability proofs are obtained by showing that the reachable set at some time instant lies entirely inside a target set.

Neural feedback systems naturally arise in safety critical systems wherein neural networks are synthesized through approaches such as reinforcement learning [42], learning from demonstrations [25] or translating a large lookup table-based controller into a more compact form using neural networks [23]. However, verification of these closed-loop systems remains a key challenge. A rapidly growing body of recent work focuses on verifying pre-/post-conditions for neural networks *in isolation* [16, 17, 26, 29, 36]. The applications to such verification are numerous, ranging from reasoning about "robustness" of classifiers used in perception systems to synthesizing adversarial inputs to improve training [5, 21, 35]. Our work considers neural networks in conjunction with ODE models.

First we note that a straightforward combination of existing tools: a flowpipe construction for ODEs [10] and a range analysis for neural networks [16] suffers from large overestimation errors due to the wrapping effect [33]. This motivates the overall approach of this paper using *rule generation*. Rather than abstract the output, our approach abstracts the function computed by the network using a local polynomial approximation along with rigorous error bounds. Formally, given a set of inputs, we compute the output as a polynomial over the input using regression. Next, we compute an error interval that conservatively accounts for the difference between the network function and the polynomial approximation. This yields a "local" Taylor model (polynomial + interval) overapproximation of the neural network that is integrated into a Taylor model-based flowpipe construction tool Flow* [7, 10]. The result is significantly less prone to runaway overestimation errors due to the wrapping effect, as shown by our evaluation.

The key technical challenge therefore lies in computing the error between a neural network and a polynomial approximation over a given range of inputs. This problem can be solved as a mixed integer nonlinear optimization (MINLP), but is significantly larger than what current MINLP solvers can handle, even for tiny neural networks. Therefore, we use an indirect approach. First, we produce a piecewise linearization (PWL) of the polynomial using branch-and-bound search with interval analysis. The error between the polynomial and the PWL approximation is guaranteed to lie within a given tolerance bound. Next, we compute the maximum and minimum difference between the neural network model and the PWL approximation using a combination of mixed integer linear programming (MILP) solver and local gradient-descent search along the lines of a recent work by Dutta et al [16]. The final error interval is obtained by adding the error between the original polynomial and the PWL model plus the error between the PWL model and the neural network.

Our experimental evaluation considers eleven neural networks that were created to stabilize a series of benchmark dynamical systems with neural network sizes ranging from 50-500 neurons and up to 6 hidden layers. The learning was carried out directly inside the Tensorflow framework [1]. Our approach is shown to be significantly faster and more accurate even with larger initial sets and a longer time horizon, when compared to a direct combination of Flow* and Sherlock tools. Thus, abstracting the function computed by the NN as opposed to just the set of outputs is necessary to avoid the wrapping effect in reachable set computation.

1.1 Related Work

The problem of constructing overapproximations to the reach sets of continuous and hybrid systems has received significant attention in the past two decades. Representative approaches for linear hybrid systems include tools such as SpaceEx [18] and HyLAA [4], while tools such as Flow* [10], CORA [2], HyCreate [3], C2E2 [14] and dReach [27] can tackle nonlinear systems. The model considered in this work consists of ODEs in feedback with neural networks that represent piecewise linear functions. Although such a model can be translated into a hybrid automaton, an upfront translation is often prohibitively expensive. An on-the-fly translation can alleviate this cost but in turn suffers from the cost of dealing with numerous mode transitions at each reachability computation step. The approach in this paper alleviates this complexity by locally approximating the feedback as a polynomial function of the inputs to the network with an appropriate error term. This avoids the need to explicitly consider mode changes in our framework.

Providing formal guarantees to neural network based feedback systems has grown in importance, since neural networks are becoming increasingly common in safety-critical applications. Given pre-condition assertions describing the inputs of a network, the verification problem asks whether the resulting outputs satisfy post-condition assertions. Numerous approaches have been proposed for neural network verification, starting from the abstraction-refinement approach of Pulina et al. [35, 36]. Therein, the nonlinear activations are systematically abstracted using an abstract relation, resulting in a linear arithmetic SMT formula. Spurious counterexamples are then used to refine the abstractions. The rapid improvements to the state-of-the-art linear arithmetic solvers such as Z3, CVC4 and MathSAT have made this approach increasingly feasible. Katz et al. present a solver specialized for neural networks with ReLU units by building on the standard simplex algorithm using special rules for handling nonlinear constraints involving the ReLU activation function [26]. Their approach was used to verify a neural network encoding advisories for an aircraft collision avoidance system. Recent work by Ehlers augments a branch-and-bound solver using facts inferred from a convexification of the activation functions [17]. Their approach can also handle max-pooling layers that are commonly used in applications in image classification. Other approaches to verification focus on the synthesis of adversarial counterexamples [5, 21] and simulation-based approaches [44].

However, the approaches mentioned above consider the network *in isolaton* which is important for a wide variety of applications. Our focus in this paper requires an approach that can propagate sets of states across networks. Lomuscio et al. evaluate an mixed integer linear program (MILP) encoding to analyze networks learned using reinforcement learning [29]. Earlier work by Dutta et al. extend the MILP approach by using local search to compute ranges over the output of a network given a polyhedron over its inputs [15]. This approach has led to a prototype tool Sherlock that produces ranges on the outputs given ranges on the inputs of the network. In principle, a combination of Sherlock with a reachability analysis tool (such as Flow*) can solve the problem at hand. However, this approach produces highly inaccurate results on all the benchmarks used in our evaluation. This happens because of the well known wrapping effect in reachability analysis. Another recent approach involves the work of Xiang et al. that computes the output ranges as a union of convex polytopes [45]. This approach does not use SMT or MILP solvers unlike other approaches and thus can lead to highly accurate estimates of the output range. However, judging from preliminary evaluation reported, the cost of manipulating polyhedra is quite expensive, and thus, the approach is currently restricted to smaller networks when compared to SMT/MILP-based approaches [15, 17, 26, 29, 36]. Another recent work by Xiang et al. considers the combination of neural networks in feedback with piecewise linear dynamical systems [46] using the techniques presented in [45]. Their approach is based on abstracting the outputs and currently lacks a detailed evaluation. Additionally, our approach handles nonlinear systems wherein the wrapping effect is often more pronounced, and thus, a bigger challenge.

Another approach to safety verification involves the use of discretized plant models and neural network controllers studied by Scheibler et al. [41] and recently by Dutta et al. [15]. These approaches use Runge-Kutta solvers to discretize the ODEs and check input/output assertions on the unrolling. Our approach here handles continuous-time dynamics specified by ODEs without requiring a discretization. To overcome the wrapping effect, our approach considers the idea of using sound rule generation. Rule generation refers to the inference of input-output relations that hold for a given set of inputs to the network [19]. The primary objective of rule generation has been to explicitly write down the “knowledge” encoded in the network in a transparent, possibly human understandable form [31]. Thus, most approaches to rule generation focus on generating a combination of Boolean implications, and are approximate in nature [43]. In this paper, we focus on rules that are of the form $y \in p(\mathbf{x}) + I$ wherein p is a polynomial over the inputs $\mathbf{x}$ to the network, y is the output of the network. Rule generation has had a long history of research in the AI community. Our approach here differs significantly in (a) the form of the rules inferred and (b) the need for *sound rule generation* with an error interval I. The use of regression in rule generation has been explored by Saito et al [38]. One key difference is that our approach includes a rigorous error

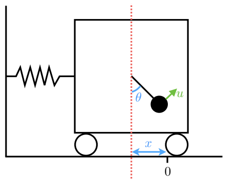


Figure 1: A diagram of the plant model for Tora

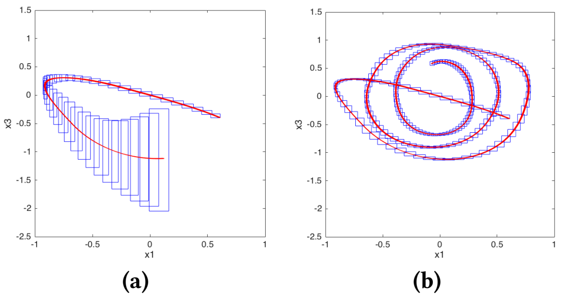


Figure 2: (a) an output range approach fails after 2 control steps due to large overestimation error, (b) polynomial rule generation approach can successfully overapproximate $N = 20$ control steps and beyond.

analysis. Also, the rules generated include a logical combination of Boolean conditions over nominal variables and polynomials.

1.2 Motivating Example

We consider the control of an electromechanical benchmark system called Tora with a rotating mass actuated by a DC motor [22], as shown in Fig. 1. The dynamics are described by a nonlinear ODE with 4 state variables and a single control input u:

$$\dot{x}_1 = x_2,\ \dot{x}_2 = -x_1 + 0.1\sin(x_3),\ \dot{x}_3 = x_4,\ \dot{x}_4 = u\,.$$

Our goal is to stabilize this system to an equilibrium state $x_i = 0$ for $i = 1, \ldots, 4$. For this purpose, we have synthesized a neural network feedback controller consisting of 3 hidden layers with a total of 300 neurons. The controller is periodic (time triggered) with a period $\tau_c = 1$ units.

For the initial condition $x_1 \in [0.6, 0.61]$, $x_2 \in [-0.7, -0.69]$, $x_3 \in [-0.4, -0.39]$ and $x_4 \in [0.59, 0.6]$, we seek to construct over approximate reach sets over a time horizon $[0, 20]$. We consider two approaches for the same: (a) *output range analysis* approach uses a combination of the tool Flow* to integrate the ODE for each control time period followed by an application of the tool Sherlock to compute the output range of the neural network. Figure 1(b) shows the resulting reachable set after $N = 2$ control steps. Unfortunately, the overapproximation error grows beyond tolerance making further flowpipe construction steps impossible. (b) the polynomial rule generation approach presented in this paper is shown in Figure 1(c). This approach is able to continue beyond $N = 20$ control steps, yielding a tight over approximation. Comparing the computed reach sets against numerical simulations shows that our approach is able to find a more accurate reachable set estimate.

2 PRELIMINARIES

In the paper, we use $\mathbb{R}$ to denote the set of all reals. A vector of variables $x_1, \ldots, x_n$ is written as $\mathbf{x}$. Its j^{th} entry is written $\mathbf{x}_j$.

Definition 2.1 (Continuous Dynamical System). A continuous dynamical system (CDS) is defined by an ODE $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}, \mathbf{w})$, wherein $\mathbf{x} \in \mathbb{R}^n$ is a vector of the state variables, $\mathbf{u} \in U$ are the control inputs, and $\mathbf{w} \in W$ are the time-varying disturbances. The sets U and W denote the bounded sets for control inputs and disturbances respectively.

The function f is assumed to be Lipschitz continuous in $\mathbf{x}$, and continuous in $\mathbf{u}$ and $\mathbf{w}$. Thus, the solution to the ODE exists for some time horizon $[0, T_{\max})$ and is unique for given control inputs and measurable disturbances.

The *evolution* under a CDS is a continuous function φ_f, also known as the forward flowmap of the ODE. Given an initial state $\mathbf{x}(0) = \mathbf{x}_0$, fixed control inputs $\mathbf{u} : [0, t] \mapsto U$ and disturbances $\mathbf{w} : [0, t] \mapsto W$, the state at some time $t \geq 0$ is given by $\mathbf{x}(t) : \varphi_f(\mathbf{x}_0, t, \mathbf{u}, \mathbf{w})$. Given an initial state set X_0, we call a state $\mathbf{x}_t$ *reachable* iff there exists some $\mathbf{x}_0 \in X_0$, $t \geq 0$, functions $\mathbf{u} : [0, t] \mapsto U$, and $\mathbf{w} : [0, t] \mapsto W$, such that the state reached at time t equals $\mathbf{x}_t$, i.e, $\varphi_f(\mathbf{x}_0, t, \mathbf{u}, \mathbf{w}) = \mathbf{x}_t$.

Definition 2.2 (Reachability Problem). The *reachability problem* for CDS has inputs (a) CDS defined by function $f(\mathbf{x}, \mathbf{u}, \mathbf{w})$, (b) a Lipschitz continuous feedback law $\mathbf{u} = g(\mathbf{x})$, wherein $g : \mathbb{R}^n \to U$, (c) disturbance set W, (d) an initial set X_0, (e) a target set X_f and (f) time horizon $[0, T]$. We ask if there exists a time trajectory of the closed-loop system starting from X_0, with disturbance signal in W that reaches the target set X_f within time $[0, T]$.

Solving reachability problems plays a key role in the safety verification of dynamical systems such that an unsafe state set is defined as a target set. However, the reachability problem on nonlinear continuous dynamics is undecidable [20]. Therefore, a common approach is to construct an overapproximation of the exact reachable set that does not intersect the unsafe set. This is supported by a variety of tools and techniques, discussed earlier in Section 1.1. Each approach is driven by a representation of sets of reachable states. The approach in this paper is built on top of Taylor models, since the dependencies of the state variables of a dynamical system can be accurately approximated by the polynomial part of a Taylor model. It further allows us to bring the dependencies from the continuous component to the discrete component in the analysis of a neural feedback system.

Taylor Model. An *interval* is the set of all reals between two bounds a, b such that $a, b \in \mathbb{R}$ and $a \leq b$. A vector interval is of the form $[\mathbf{a}, \mathbf{b}]$ for $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$ represents a Cartesian product $\prod_{j=1}^{n} [\mathbf{a}_j, \mathbf{b}_j]$. Such an interval forms a box or a hyperrectangle in $\mathbb{R}^n$. Interval arithmetic extends standard arithmetic operators from floating point numbers to intervals [32]. Taylor models are a higher-order extension of interval arithmetic.

Definition 2.3. A *Taylor Model (TM)* is denoted by a pair (p, I) wherein p is a polynomial over $\mathbf{x}$, whose domain D is an interval, and I is an interval.

Given a function $f(\mathbf{x})$ with $\mathbf{x} \in D$, a TM (p, I) overapproximates f if and only if $f(\mathbf{z}) \in p(\mathbf{z}) + I$ for all $\mathbf{z} \in D$. We also call (p, I) a

TM of f. TMs can also be organized as vectors to overapproximate vector-valued functions.

TMs are closed under most basic arithmetic operations and the overapproximation property is conserved. For example, assume that (p_f, I_f) and (p_g, I_g) are TMs of the functions f and g respectively over the domain D, then the summation $(p_f, I_f) + (p_g, I_g) = (p_f + p_g, I_f + I_g)$ is a TM of $f + g$. Other operations include multiplication, application of any smooth function, differentiation and integration. TM arithmetic was originally developed by Berz and Makino (see [6, 30]), and a powerful integration technique which is called *TM integration* [7, 9] is implemented based on it. The extension of the method also allows ODEs to have time-varying disturbances [8].

Given an ODE $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}, \mathbf{w})$ with a feedback law $\mathbf{u}(t) = g(\mathbf{x}, t)$, a time horizon $[0, T]$ and integration time step τ_I, we may use a TM integrator such as Flow* to compute a series of N TMs $(p_0, I_0), \ldots, (p_{N-1}, I_{N-1})$ wherein $N = \lceil \frac{T}{\tau_I} \rceil$, and (p_j, I_j) is a TM that overapproximates the reachable sets of the closed loop system over the time interval $[j\tau_I, (j+1)\tau_I]$. The tool may also adaptively vary the integration time step τ_I and the degree of the polynomials p_j using heuristics that are described elsewhere [8].

Neural Network. Next, we define feedforward neural networks (FNNs). Structurally, a FNN $\mathcal{N}$ consists of $k > 0$ hidden layers, wherein we assume that each layer has the same number of neurons $N > 0$. We use N_{ij} to denote the j^{th} neuron of the i^{th} layer for $j \in \{1, \ldots, N\}$ and $i \in \{1, \ldots, k\}$.

Definition 2.4 (Neural Network). A k layer, n input , neural network with N neurons per hidden layer is described by matrices: $(W_0, \mathbf{b}_0), \ldots, (W_{k-1}, \mathbf{b}_{k-1}), (W_k, \mathbf{b}_k)$, wherein (a) $W_0, \mathbf{b}_0$ are $N \times n$ and $N \times 1$ matrices denoting the weights connecting the inputs to the first hidden layer, (b) $W_i, \mathbf{b}_i$ for $i \in [1, k-1]$ connect layer i to layer $i+1$ and (c) $W_k, \mathbf{b}_k$ connect the last layer k to the output.

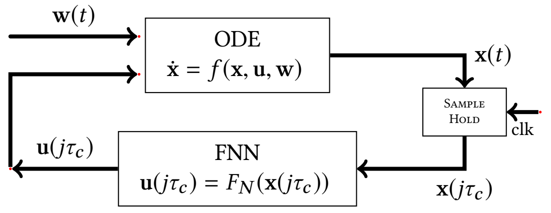
Each neuron is defined using its *activation function* σ linking its input value to the output value. Although this can be any nonlinear function, we focus on neural networks with "ReLU" activation function $\sigma(z) : \max(z, 0)$. However, the techniques presented in this paper extend to other types of activation units through piecewise linearization [16].

For a neural network $\mathcal{N}$, as described above, the function $F_N : \mathbb{R}^n \to \mathbb{R}$ computed by the neural network is given by the composition $F_N := F_k \circ \cdots \circ F_0$ wherein $F_i(\mathbf{z}) : \sigma(W_i\mathbf{z} + \mathbf{b}_i)$ is the function computed by the i^{th} hidden layer, F_0 the function linking the inputs to the first layer, and F_k linking the last layer to the output. Note that the function defined by a neural network with ReLU activation functions is continuous and piecewise differentiable.

Range Analysis for Neural Networks. The problem of range analysis for a neural network starts from a network $\mathcal{N}$ and a set $\mathbf{x} \in D$ of inputs to the network. The goal is to find an interval $[\ell, u]$ such that $(\forall\ \mathbf{x} \in I)\ F_N(\mathbf{x}) \in [\ell, u]$.

Often, we are interested in ensuring that the interval is tight. Finding such an interval over the outputs is performed by solving two optimization problems:

$$\max(\min)\ y \text{ s.t } \mathbf{x} \in I\ \wedge\ y = F_N(\mathbf{x}),$$



$\mathbf{w}(t)$
ODE
$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}, \mathbf{w})$
$\mathbf{x}(t)$
Sample Hold
clk
$\mathbf{u}(j\tau_c)$
FNN
$\mathbf{u}(j\tau_c) = F_N(\mathbf{x}(j\tau_c))$
$\mathbf{x}(j\tau_c)$

Figure 3: Block diagram of a neural feedback control system.

However, the problem of solving optimization problems with neural network constraints is highly nonlinear. Using the properties of ReLU function, it can be encoded as a large mixed integer linear program (MILP) [16, 29]. Recent work by Dutta et al, augments the MILP approach by using local gradient information to improve the current solution. While the approach uses an MILP solver to perform global search, it is only asked to provide a small ϵ improvement to an existing local solution, when it is stuck in a local minima. The combined approach is reported to be faster and more effective for many of the networks tested, and implemented inside the tool Sherlock [16].

3 PROBLEM STATEMENT AND APPROACH

We present the problem statement and a high level overview of our approach.

3.1 Problem Statement

Definition 3.1 (Neural Feedback System). A *Neural Feedback System* $\mathcal{S}$ is a tuple $\langle X, U, W, f(\mathbf{x}, \mathbf{u}, \mathbf{w}), \mathcal{N}, \tau_c \rangle$ wherein $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}, \mathbf{w})$ defines the dynamics of the continuous component, $X \subseteq \mathbb{R}^n$ is the state space of the system, $U \subseteq \mathbb{R}^m$ is the control input range, and $W \subseteq \mathbb{R}^l$ is the range of the time-varying disturbances. Finally, τ_c is the time period of the controller, i.e., the control stepsize.

Figure 3 shows a block diagram representation of a NFS. The feedback $\mathcal{N}$ is a FNN with input $\mathbf{x} \in X$ and yields an output $\mathbf{u} \in U$. The neural network is invoked at time instants $t = j\tau_c$ for $j \in \mathbb{N}$, with the output of the network held constant over times $t \in [j\tau_c, (j+1)\tau_c)$. The network is assumed to compute its output instantaneously whenever its inputs change.

Given a bounded time horizon $[0, T]$, initial state $\mathbf{x}_0$ and a disturbance $\mathbf{w} : [0, T] \mapsto W$, trajectory $\mathbf{x}(t)$ and control signal $\mathbf{u}(t)$ for $t \in [0, T]$ are defined as follows. For each time interval $t \in [j\tau_c, (j+1)\tau_c]$ such that $j = 0, 1, \ldots, \frac{T}{\tau_c} - 1$, we have that $\mathbf{x}(t) = \varphi_f(\mathbf{x}(j\tau_c), t - j\tau_c, \mathbf{u}(t), \mathbf{w}(t))$ and $\mathbf{u}(t) = F_N(\mathbf{x}(j\tau_c))$.

It is obvious that the reachability problem is undecidable on NFSs, since it is already undecidable on CDS. Thus we want to compute an accurate overapproximation for the reachable set of a NFS in order to prove its safety.

3.2 Our Approach

Our approach exploits the local continuity properties of the feedback function $F_N(\mathbf{x})$. Rather than consider the given NFS as a hybrid automaton, we will consider it as a continuous feedback system and locally approximate the feedback F_N by a polynomial of a given degree, while carefully accounting for the error.

Given an initial set X_0 and a reachability computation task for N control steps, spanning a time horizon $T = N\tau_c$. Our approach uses an integration step $\tau_I = \frac{\tau_c}{M}$, wherein M flowpipes are constructed for each control step or time period. Therefore, we successively produce a sequence of TM flowpipes of a fixed order $k > 0$:

$$\underbrace{R_{1,1}, \ldots, R_{1,M}}_{\text{Ctrl. Step \# 1}}, \;\cdots\; \underbrace{R_{N,1}, \ldots, R_{N,M}}_{\text{Ctrl. Step \# } N}.$$

such that for each $j = 1, \ldots, N$ and $i = 1, \ldots, M$, $R_{j,i}$ is an overapproximation of the reachable set in the time interval of $[(j-1)\tau_c + (i-1)\tau_I, (j-1)\tau_c + i\tau_I]$. More intuitively, the reachable set in each control step is overapproximated by M TM flowpipes.

For the j^{th} control step such that $j = 1, \ldots, N$, our algorithm performs the following steps.

(1) Compute the order k TM overapproximation X_j for the reachable set at time $t = (j-1)\tau_c$.
(2) We compute a *polynomial rule* $q_j(\mathbf{x})$ as well as an error interval J_j which are valid for any input $\mathbf{x} \in X_j$ of the FNN controller, i.e., $(\forall \mathbf{x} \in X_j).(F_N(\mathbf{x}) \in q_j(\mathbf{x}) + J_j)$ wherein F_N denotes the input/output mapping of the FNN. We call this step *rule generation*, and discuss this in the subsequent sections.
(3) Compute the control input $\mathbf{u}_j = q_j(X_j) + J_j$ for the current control step by TM arithmetic with the order k.
(4) Update the continuous dynamics to $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}_j, \mathbf{w})$, and perform TM integration with the stepsize $\tau_I : \frac{\tau_c}{M}$ to compute the order k TM flowpipes $R_{j,1}, \ldots, R_{j,M}$ for the current control step. Then the new flowpipes are appended to the resulting list.

By doing so, the dependencies among the state variables in the system evolution can be transferred between the continuous and discrete components, so that the overall overestimation in reachability computation can be greatly reduced. In the next section, we will describe the rule generation step in detail.

Remark. A more direct approach could be constructing a hybrid automaton $\mathcal{A}$ on the fly for the executions of the given NFS $\mathcal{S}$, and then perform the safety verification on $\mathcal{A}$ to prove the safety of $\mathcal{S}$. However, such a method requires to introduce a discrete mode to $\mathcal{A}$ each time a linear region in the FNN is visited in a computation path, and the total number of linear regions is exponential in the number of neurons in the FNN. As we will show in Section 7 that, in each test, the number of our piecewise linear sections is much smaller than the number of linear regions of the FNN controller. Such an approach would also lead to intersections of flowpipes with hyperplanes and a loss in precision as a result. Our approach here avoids direct intersections between TMs and guards.

4 POLYNOMIAL RULE GENERATION

In this section, we will describe the polynomial rule generation problem and a rigorous rule generation approach. Let $\mathcal{N}$ be a neural network with n inputs written as $\mathbf{x} \in \mathbb{R}^n$ and a single output $y \in \mathbb{R}$. Let D be a given domain of the inputs $\mathbf{x}$. The purpose of the rule generation is to keep the dependencies of the variables under the FNN input/output mapping as much as possible.

Definition 4.1 (Polynomial Rule Generation Problem). The inputs to the *polynomial rule generation* problem include (a) neural network $\mathcal{N}$, (b) input domain D and (c) desired order of the TM k. The output is a TM (p, I), known in this context as a *polynomial rule* for the network, such that $(\forall\ \mathbf{x} \in D)\ F_N(\mathbf{x}) \in p(\mathbf{x}) + I$, i.e., $p(\mathbf{x}) + I$ is an overapproximation of $F_N(\mathbf{x})$ w.r.t. $\mathbf{x} \in D$.

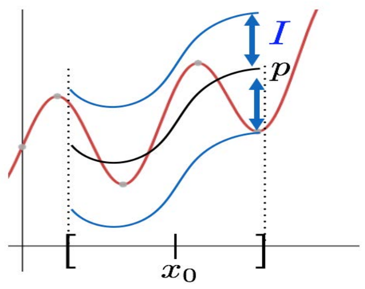


Figure 4: Polynomial Rule plus Interval: The red curve shows the actual behavior of the Neural Network Controller, around the point x_0. The black curve shows the polynomial obtained by regression, and the blue curves show the upper and lower bound polynomials after adding the interval error I to the polynomial p

Our overall approach to polynomial rule generation has two main steps:
(a) We use polynomial regression over sample input/output pairs $(\mathbf{x}_i, y_i)_{i=1}^K$ obtained by sampling the domain D and computing $y_i = F_N(\mathbf{x}_i)$ for each sample. The result of the polynomial regression is the polynomial p.
(b) We estimate an interval I that subsumes the range of the error $e(\mathbf{x}) : F_N(\mathbf{x}) - p(\mathbf{x})$, that is, to ensure the overapproximation property. We would refer the reader to Fig 4 as an illustration of this approach.

4.1 Polynomial Regression

The first step in our rule generation approach is to compute a polynomial $p(\mathbf{x})$ through regression. To do so, we generate samples from the domain D. Let $\mathbf{x}_1, \ldots, \mathbf{x}_K \in D$ denote the samples thus obtained. The outputs $y_i : F_N(\mathbf{x}_i)$ are computed by evaluating the neural network over the obtained samples.

Next, given the desired order k, let N_k denote all vectors $\alpha \in \mathbb{N}^n$ of size n over natural numbers, such that $\sum_{i=1}^n \alpha_i \leq k$. We write $\mathbf{x}^\alpha$ as a shortcut for the monomial $\prod_{i=1}^n \mathbf{x}_i^{\alpha_i}$. A generic polynomial template of order upto k is written as $p(\mathbf{x}; \mathbf{c}) : \sum_{\alpha \in N_k} \mathbf{c}_\alpha \mathbf{x}^\alpha$.

The goal of least squares regression is to find values of the coefficients $\mathbf{c}$ such that the sum of square of the error for each sample $\mathbf{x}_i$ is minimized:

$$\min_{\mathbf{c}} \sum_{j=1}^{K} (y_i - p(\mathbf{x}_i; \mathbf{c}))^2 .$$

This can be solved readily as a linear Ordinary Least Squares (OLS) problem by constructing a matrix M whose rows range from $i = 1, \ldots, K$ wherein the ith row represents the sample $\mathbf{x}_i$. The columns of M range over the polynomials $\mathbf{x}^\alpha$ for monomials $\alpha \in N_k$. Once M is constructed, we solve the least squares problem $M\mathbf{c} \simeq \mathbf{y}$ using off-the-shelf approaches available in most linear algebra packages.

However, in many instances, OLS approaches to polynomial regression yield large coefficients $\mathbf{c}$ that make the error computation quite expensive. To control the size of the coefficients, we adopt two popular ideas: (a) We use *ridge regression*, wherein we add the norm of $\mathbf{c}$ as a penalty function to the objective.

$$\min_{\mathbf{c}} \sum_{j=1}^{K} (y_i - p(\mathbf{x}_i;\mathbf{c}))^2 + \gamma \mathbf{c}^T \mathbf{c}\,.$$

Here γ is a constant that weights the penalty term with respect to the regression error. (b) Rather than construct an order k model in "one shot", we start by first fitting an order 1 (linear model) $p_1(\mathbf{x})$. Next, we fit a purely quadratic model to the residual function $y_i - p_1(\mathbf{x}_i)$. The result is an order two model $p_1 + p_2$. We proceed thus until the maximum residual is within tolerance. This approach provides yet another way to bias the search towards lower degree polynomials.

4.2 Error Analysis

Next, we will focus on computing the error between a polynomial $p(\mathbf{x})$ and the given network $\mathcal{N}$ over a domain D. Let $e(\mathbf{x}) : F_N(\mathbf{x}) - p(\mathbf{x})$ denote the difference between the neural network output and the polynomial $p(\mathbf{x})$. Therefore, we seek to compute an interval $I : [a, b]$ such that $a \leq \min_{\mathbf{x}\in D} e(\mathbf{x}) \leq \max_{\mathbf{x}\in D} e(\mathbf{x}) \leq b$. Furthermore, we wish our bounds to be "tight", in practice.

However, finding the optimum value of $e(\mathbf{x})$ over D is a large mixed-integer nonlinear optimization problem, which is quite expensive to solve in practice. Since our goal is to overapproximate the range of e, we proceed in two steps: (a) We approximate p using piecewise linear models (PWL). (b) We compute the error between the PWL models and the neural network.

Each of the steps is described in the subsequent sections.

5 FROM POLYNOMIALS TO PIECEWISE LINEAR MODELS

In this section, we describe the approximation of a given polynomial $p(\mathbf{x})$ over a domain D by piecewise linear (PWL) models.

Definition 5.1 (PWL Function). A PWL function $f : D \mapsto \mathbb{R}$ over a domain D is a set of *linear pieces* $(R_j, \mathbf{c}_j, d_j)_{j=1}^{M}$ such that (a) each $R_j \subseteq D$ is a hyper-rectangle; (b) the union of rectangles cover D: $\bigcup_{j=1}^{n} R_j = D$; and (c) $R_i \cap R_j = \emptyset$ for $i \neq j$. The function f is defined as $f(\mathbf{x}) :\ \mathbf{c}_j\mathbf{x} + d_j$ whenever $\mathbf{x} \in R_j$.

Although we have defined a PWL function over non-intersecting examples: our representation of these functions used subsequently will perform a topological closure to allow rectangles to share common faces. The result is technically a PWL relation rather than a function. Given a domain D, a polynomial $p(\mathbf{x})$ and a desired tolerance $\epsilon > 0$, we seek to find a PWL approximation $f : D \mapsto \mathbb{R}$ s.t. $(\forall\ \mathbf{x} \in D)\ |f(\mathbf{x}) - p(\mathbf{x})| \leq \epsilon$.

Algorithm 1 shows the overall scheme to systematically construct a PWL model for a polynomial with a given error tolerance $[-\epsilon, \epsilon]$. The parameter $\delta > 0$ is used primarily by the **FindMax-Interval** procedure. The algorithm maintains a set S, which is a union of mutually disjoint hyperrectangles. At each iteration of the loop (line 4), it finds a point $\mathbf{x}_s \in S$ and constructs a linearization f_s around $\mathbf{x}_s$ (line 7). It then uses the method **FindMaxInterval** to estimate an interval B_s around $\mathbf{x}_s$ such that the $|p(\mathbf{x}) - f_s(\mathbf{x})| \leq \epsilon$ for all $\mathbf{x} \in B_s$. The region B_s is removed from further consideration (line 10) and a linear piece is added to the PWL model L.

Algorithm 1: Algorithm to systematically compute PWL model by selecting a new sample and building a maximal interval around it, given polynomial $p(\mathbf{x})$ over domain D with tolerance ϵ and minimum box width δ.

```
1:  procedure FindPWLApproximation($p$, $D$, $\epsilon$, $\delta$)
2:      $S \leftarrow D$;  ▷ $S \subseteq D$ represents the region that remains to be examined.
3:      $L \leftarrow \emptyset$;  ▷ $L$ represents the set of linear pieces thus far.
4:      while $S \neq \emptyset$ do
5:          $\mathbf{x}_s \leftarrow$ getSample($S$);  ▷ Get a current sample from $S$.
6:          $(\mathbf{c}_s, d_s) \leftarrow (\nabla\, p(\mathbf{x}_s),\ p(\mathbf{x}_s))$;
7:          $f_s : \lambda\mathbf{x}.\ \mathbf{c}_s^T(\mathbf{x} - \mathbf{x}_s) + d_s$;  ▷ Compute linearization around $\mathbf{x}_s$
8:          $B_s \leftarrow$ FindMaxInterval($\mathbf{x}_s, p - f_s, \epsilon, \delta, S$);  ▷ Compute interval $B_s$.
9:                                  ▷ FindMaxInterval guarantees that $(\forall\ \mathbf{x} \in B_s)\ |p(\mathbf{x}) - f_s(\mathbf{x})| \leq \epsilon$.
10:         $S \leftarrow S \setminus B_s$;
11:         $L \leftarrow L \cup \{(B_s,\ \mathbf{c}_s,\ d_s - \mathbf{c}_s^T\mathbf{x}_s)\}$; ▷ Add to PWL model.
12:     return $L$.  ▷ return the final PWL model
```

The algorithm relies on the routine **FindMaxInterval**. This routine is shown in Algorithm 2. This routine attempts to find a box B around the current sample $\mathbf{x}$ such that the range of a given polynomial f inside B lies within $[-\epsilon, \epsilon]$. The approach first builds a box of width δ around the given sample $\mathbf{x}$ (line 2). If the range of the function inside this box fails to be within the given tolerance, we conclude that the minimum box width is too large with respect to the desired tolerance ϵ and terminate with failure (line 5). Otherwise, the approach attempts a series of box expansions. The symbol $\ll_i$ is used to denote a reduction of the current lower bound for x_i by δ (line 11), whereas $\gg_i$ denotes an increase to the current upper bound by δ (line 12). If the change to the interval requested by current symbol succeeds in that the new interval continues to lie within S (line 14) and the range of f continues to lie within $[-\epsilon, \epsilon]$ (lines 16, 17), we update the current box (line 19) and save the current symbol (line 20). Otherwise, we discard the change and remove the current symbol from future consideration.

Algorithm 2 relies on the routine **EvaluateRange** that returns an interval J that overapproximates the range of a polynomial f over an interval I. We assume that the procedure **EvaluateRange** is sound: $J \supseteq \{y \mid y = f(\mathbf{x}),\ \mathbf{x} \in I\}$.

Theorem 5.2. *For any polynomial f, sample $\mathbf{x}$, set S, tolerance ϵ and minimum box width δ, the* ***FindMaxInterval*** *routine (a) always terminates; (b) if it succeeds, returns a box B such that $f(B) \subseteq [-\epsilon, \epsilon]$.*

A proof is provided in the appendix. Successful execution of algorithm 2 requires us to implement a sound range evaluation procedure **EvaluateRange** and choose values of ϵ, δ so that the assertion in line 5 always succeeds.

Algorithm 2: For a given polynomial function $f(\mathbf{x})$, find largest interval B around sample $\mathbf{x}$ such that $B \subseteq S$ and $|f(\mathbf{x})| \leq \epsilon$. The input δ is the smallest allowable interval.

```
procedure FindMaxInterval($\mathbf{x}, f, \epsilon, \delta, S$)
    $[\mathbf{a}, \mathbf{b}] \leftarrow [\mathbf{x} - \frac{\delta}{2}\mathbf{1}, \mathbf{x} + \frac{\delta}{2}\mathbf{1}]$;    ▷ Form initial box around x
    $J_0 \leftarrow$ EvaluateRange($f, [\mathbf{a}, \mathbf{b}]$);
                                    ▷ Compute range of f over initial box.
    ASSERT( $J_0 \subseteq [-\epsilon, \epsilon]$);
                        ▷ Failure: either ϵ is too small or δ is too large.
    wlist $\leftarrow \{\ll_1, \ldots, \ll_n, \gg_1, \ldots, \gg_n\}$;
    ▷ $\ll_j$: decrease lower bound $x_j$ and $\gg_j$: increase the upper bound for $x_j$
    while wlist $\neq \emptyset$ do
        $s \leftarrow$ pop(wlist);        ▷ pop from the worklist of actions.
        if $s = \ll_i$ then $\hat{\mathbf{a}} \leftarrow \mathbf{a} - \delta\mathbf{e}_i$, $\hat{\mathbf{b}} = \mathbf{b}$; end if
        if $s = \gg_j$ then $\hat{\mathbf{a}} \leftarrow \mathbf{a}$, $\hat{\mathbf{b}} = \mathbf{b} + \delta\mathbf{e}_j$; end if
        if $[\hat{\mathbf{a}}, \hat{\mathbf{b}}] \subseteq S$ then
                            ▷ Ensure that new box remains inside S
            $J \leftarrow$ EvaluateRange($f, [\hat{\mathbf{a}}, \hat{\mathbf{b}}]$);
                                            ▷ evaluate range of f
            if $J \subseteq [-\epsilon, \epsilon]$ then
                                    ▷ error remains within tolerance?
                $\mathbf{a} \leftarrow \hat{\mathbf{a}}$, $\mathbf{b} \leftarrow \hat{\mathbf{b}}$;       ▷ update the current box
                push(wlist, $s$);
                                ▷ save current direction to try again
    return $[\mathbf{a}, \mathbf{b}]$;
```

Lemma 5.3. *Algorithm 2 is always called with a function f and $\mathbf{x}$ such that $f(\mathbf{x}) = 0$ and $\nabla f(\mathbf{x}) = 0$. Furthermore $S \subseteq D$.*

The proof is simply to examine the arguments at the only call site to **FindMaxInterval** in Algorithm 1.

Theorem 5.4. *For any compact set D, and fixed tolerance $\epsilon > 0$, there is a sound procedure **EvaluateRange** and a corresponding value of δ such that the assertion check in line 5 of Algorithm 1 always succeeds.*

The explicit formula for setting δ is provided in the appendix.

Using the properties of the FindMaxInterval method, we now provide guarantees for Algorithm 1.

Theorem 5.5. *If Algorithm 1 terminates with success for input p over domain D with tolerance ϵ, then the resulting set of linear pieces L define a PWL function f_L such that $|f_L(\mathbf{x}) - p(\mathbf{x})| \leq \epsilon, \ \forall\, \mathbf{x} \in D$.*

Data Structures: We note that Algorithms 1 and 2 rely on a data structure that maintains the disjoint union of boxes. Furthermore, Alg. 2 guarantees that the corners of these box lie on a uniform grid of size δ along each axis of the original domain D.

We use a modification of the standard *kd*-tree data structure to carry out the basic operations that include (a) find an cell in S and return its center point; (b) check if a box lies entirely inside S; and (c) remove a box from S [39]. The details of this data structure and its implementation will be discussed in an extended version.

Decomposed PWL Models: Another significant detail is that when the dimensionality of the space is large, the approach of gridding the state space can cause the number of linear pieces to explode, making it prohibitively expensive in practice. As a result, we exploit the fact that p is generally of low degree and is often sparse due to the nature of the regression techniques used to construct it.

Therefore, we write $p(\mathbf{x})$ as a sum of polynomials, each involving a much smaller number of variables:

$$p(\mathbf{x}) : \ p_1(\mathbf{x}_{1,1}, \ldots, \mathbf{x}_{1,k}) + \cdots + p_J(\mathbf{x}_{J,1}, \ldots, \mathbf{x}_{J,k}).$$

More specifically, each of the summands need involve at most k out of the n variables, where k is the order of p. Therefore, our approach separately considers PWL models for $p_1, \ldots, p_J$ with tolerance $\frac{\epsilon}{J}$. In practice, since k is typically 2 or 3, we are able to construct PWL models through subdivisions without suffering from the curse of dimensionality.

EvaluateRange Procedure: Theorem 5.4 (proof in Appendix) constructs a sound **EvaluateRange** procedure along with a value of δ so that the assertion failure in Line 5 of Algorithm 2 never happens. This is quite cumbersome to implement, in practice. Our implementation uses standard affine arithmetic evaluation [13] built on top of the MPFI interval arithmetic library [37].

Setting Parameters: Line 5 of Algorithm 2 has an assertion that requires the user to set parameters ϵ, δ in the right combination to avoid an assertion failure. In practice, this is quite cumbersome. Therefore, we allow the user to set ϵ, δ initially. If the condition in line 5 is not satisfied, we increase ϵ to force it to be satisfied. Note that in doing so, the linear pieces already constructed in Algorithm 1 are not invalidated since they satisfy a smaller tolerance. Also, our implementation allows the user to specify a different value of δ along each dimension of $\mathbf{x}$.

6 ERROR ANALYSIS USING OPTIMIZATION

In the previous section, we showed how a polynomial $p(\mathbf{x})$ over a domain D can be replaced using a piecewise linear function $f(\mathbf{x})$ such that for all $\mathbf{x} \in D$, $|p(\mathbf{x}) - f(\mathbf{x})| \leq \epsilon$, for a given $\epsilon > 0$. In this section, we complete the rule generation for a given neural network $\mathcal{N}$ by computing bounds on $|f_N(\mathbf{x}) - f(\mathbf{x})|$ over $\mathbf{x} \in D$. Thus,

$$|f_N(\mathbf{x}) - p(\mathbf{x})| \leq |f_N - f| + \underbrace{|f - p|}_{\leq \epsilon}.$$

Our approach builds on earlier work on output range generation of neural networks, wherein we pose the problem as a mixed integer linear program (MILP), and next combine local search with MILP solvers to yield more efficient bounds estimation.

Definition 6.1 (Neural Network to PWL Error). Given a neural network $\mathcal{N}$ over inputs $\mathbf{x} \in D$ and a PWL model $f_L : D \mapsto \mathbb{R}$, find an interval $[a, b]$ such that $(\forall\, \mathbf{x} \in D)\ (f_N(\mathbf{x}) - f_L(\mathbf{x})) \in [a, b]$.

To do so, we will first define a mixed integer linear programming (MILP) by separately encoding the network $\mathcal{N}$ and the PWL model L into MIL constraints.

Definition 6.2 (Mixed Integer LP (MILP)). Let $\mathbf{x} \in \mathbb{R}^n$ be a set of real variables and $\mathbf{v} \in \mathbb{Z}^m$ be a set of integer variables. A MILP over $\mathbf{x}, \mathbf{v}$ is an optimization problem of the form:

$$\max\ \mathbf{a}_x^T \mathbf{x} + \mathbf{a}_w^T \mathbf{v} \text{ s.t. } A\mathbf{x} + B\mathbf{v} \leq \ \mathbf{c}\,.$$

First, given a neural network $\mathcal{N}$ over inputs $\mathbf{x} \in D$ and output $y \in \mathbb{R}$, we derive a set of linear constraints $\Psi_N(\mathbf{x}, y, \mathbf{v})$ over $(\mathbf{x}, y) \in \mathbb{R}^{n+1}$ and binary variables $\mathbf{v} \in \{0,1\}^M$, such that for any $\mathbf{x} \in D$, if $z = F_N(\mathbf{x})$ then $(\exists\ \mathbf{v} \in \{0,1\}^M)\ \Psi_N(\mathbf{x}, z, \mathbf{v})$. In other words, the constraints Ψ capture all possible input output pairs for the network $\mathcal{N}$. Conversely, if the network is constructed using ReLU units, we conclude that whenever $(\exists\ \mathbf{v} \in \{0,1\}^M)\ \Psi_N(\mathbf{x}, z, \mathbf{v})$ holds, we have $z = F_N(\mathbf{x})$. This encoding is described in detail elsewhere [16, 29].

Encoding PWL Models: Let f_L be a PWL model defined by $L : \langle R_j, \mathbf{c}_j, d_j \rangle_{j=1}^N$, over $\mathbf{x} \in D$ (see Def. 5.1). Let D be represented by the interval $[\mathbf{a}_D, \mathbf{b}_D]$. We briefly describe the compilation of the PWL model into constraints. To do so, we use variables $\mathbf{x} \in \mathbb{R}^n$ for the inputs and $z \in \mathbb{R}$ for the output of the model. Additionally, we will introduce a fresh binary variable $l_j \in \{0,1\}$ corresponding to the piece $\langle R_j, \mathbf{c}_j, d_j \rangle$. The first constraint encodes that only one of the pieces can apply.

$$l_1 + l_2 + \cdots + l_N = 1 \tag{6.1}$$

Next, we note that if $l_i = 1$, then $\mathbf{x} \in R_j$. Let $R_j : [\mathbf{a}_j, \mathbf{b}_j]$.

$$\mathbf{a}_j l_j + \mathbf{a}_D(1 - l_j) \le \mathbf{x} \le \mathbf{b}_j l_j + \mathbf{b}_D(1 - l_j)\,. \tag{6.2}$$

Next, we need to encode the relation between the output z and inputs $\mathbf{x}$ whenever piece j is selected. To this effect, let M be a large constant chosen so that for all $\mathbf{x} \in D$, (a) $f_L(\mathbf{x}) \in [-M, M]$, and (b) $|\mathbf{c}_j^T\mathbf{x} + d_j| \le M$ for $j = 1, \ldots, N$. We can encode the input output relation for the PWL as follows:

$$\mathbf{c}_j^T\mathbf{x} + d_j - 2M(1 - l_j) \le z \le \mathbf{c}_j^T\mathbf{x} + d_j + 2M(1 - l_j) \tag{6.3}$$

The overall MIL constraints are given as $\Psi_L(\mathbf{x}, z, \vec{l})$ taken as the conjunction of (6.1), (6.2) and (6.3), wherein $\vec{l} : (l_1, \ldots, l_N)$ collects the binary variables. The MILP encoding precisely captures the function represented by the PWL model.

Theorem 6.3. *For all $\mathbf{x} \in D$, If $z = f_L(\mathbf{x})$ then, $(\exists\ \vec{l} \in \{0,1\}^N)$ $\Psi_L(\mathbf{x}, z, \vec{l})$.*

The converse will also hold in general, if our encoding did not effectively compute the topological closure of each rectangle in L. Ensuring this will yield MILPs with strict inequalities, and therefore is omitted for simplicity of presentation.

Combined MILP Model: Given the constraints $\Psi_N(\mathbf{x}, y, \mathbf{v})$ for a neural network N and constraints $\Psi_L(\mathbf{x}, z, \vec{l})$ for a PWL model L, the error interval is estimated by setting up a two MILPs as follows:

$$\begin{array}{lll}
\max(\min) & z - y & \\
\text{s.t.} & \Psi_N(\mathbf{x}, y, \mathbf{v}) & \text{(*MILP encoding for NN*)} \\
& \Psi_L(\mathbf{x}, z, \vec{l}) & \text{(*MILP encoding for PWL*)} \\
& \mathbf{x} \in D,\ (\mathbf{v}, \vec{l}) \in \{0,1\}^{|\mathbf{v}| + |\vec{l}|} &
\end{array}$$

It is clear that the solutions to the MILP problem above yields the required error bound between the PWL model and the neural network. Combining this with the tolerance between the polynomial $p(\mathbf{x})$ and the PWL model yields the total error interval.

Theorem 6.4. *The PWL model along with the error interval computed above overapproximate the range of $f_N(\mathbf{x})$ wherein $\mathbf{x} \in D$.*

7 EXPERIMENTAL RESULTS

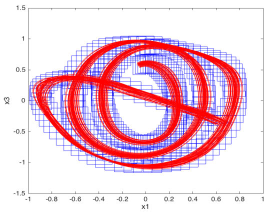


Figure 5: Flowpipes for the Tora example with a larger initial set

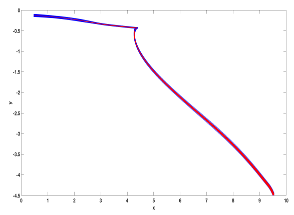


Figure 6: Flowpipes for the Car Model

We implemented a prototype tool for our rule generation as well as error analysis techniques and use it together with the tool Flow* and Sherlock. The TM flowpipes under continuous dynamics are computed by Flow* with the symbolic remainder technique described in [11]. The polynomial rule generation procedure described in Algorithms 1 and 2 along with the MILP encoding were implemented on top of the tool Sherlock. The experiments were run on a MacBook Pro Laptop, with 2.7 GHz Intel Core i5, with 16 GB RAM. The source code for repeating our experiments, can be found at *bit.ly/2Ibhfha* . The virtual machine with all the dependencies set up, and running experiments can be obtained by requesting the authors.

Benchmarks. We consider the continuous dynamical systems described in [24, 28, 34, 40, 47], and create the NFS benchmarks given in Table 1. For each system, the controller is a neural network which is trained using a standard MPC control scheme. Each benchmark is also equipped with a time-varying disturbance which is added to the control input. Our purpose is to prove that for each system, all state variables stay in the safe range of $[-2, 2]$ during the first N control steps from the initial set.

The benchmark #9 is our motivating example while with a much larger initial set. A sample reach set computation for 0.1 seconds of Benchmark 9 has been shown here. We start with a set given by the interval : $I = [0.6, 0.7] \times [-0.7, -0.6] \times [-0.4, -0.3] \times [0.5, 0.6]$. By uniformly sampling interval I we obtain the following polynomial, through regression:

$$\begin{aligned}
p(x_0, x_1, x_2, x_3) = & 0.62 + 1.01x_0 + 0.54x_1 - 0.69x_2 - 2.1x_3 \\
& + 3.1\text{e-}4x_0^2 + 7.1\text{e-}4x_0x_1 + 1.5\text{e-}4x_1^2 \\
& + 1.1\text{e-}4x_0x_2 - 1.6\text{e-}4x_1x_2 + 1.5\text{e-}4x_2^2 \\
& - 2.5\text{e-}4x_0x_3 - 6.5\text{e-}4x_1x_3 + 6.8\text{e-}5x_2x_3 + 5.5\text{e-}5x_3^2
\end{aligned}$$

The max error between the neural network, and p, in the domain I is deduced as $e = 0.0178211$. That is, the neural network behavior is overapproximated by the TM : $p(x_0, x_1, x_2, x_3) + [-e, e]$. Using this TM as the feedback function, the flowpipe computed yields the following set, after $0.1s$ of time, $[0.53, 0.63] \times [-0.77, -0.66] \times [-0.35, -0.24] \times [0.49, 0.60]$.

Results. We present our experimental results in Table 2. We use the regression order 2 in all of our tests, and to provide a comparison, we give the column T_I for the time costs of a direct combination of Flow* and Sherlock, although it works on none of our benchmarks. In all of the tests, we use the symbolic remainder method provided

Table 1: Suite of benchmarks used for testing the proposed method. Legend : Var: # of state variables, N : # of control steps for computing the reach sets , τ_c : Duration of control time steps, NN :, Neural Network k : # of layers in the Neural Network, N : # of neurons, $init$: Initial set for reachability computation, w : Disturbance range.

#	Var	N	τ_c	NN k	NN N	$init$	w
1	2	30	0.2	5	56	$[0.5, 0.9]^2$	$\pm 10^{-2}$
2	2	50	0.2	6	156	$[0.7, 0.9] \times [0.42, 0.58]$	$\pm 10^{-3}$
3	2	100	0.1	5	56	$[0.8, 0.9] \times [0.4, 0.5]$	$\pm 10^{-2}$
4	3	50	0.2	6	156	$[0.35, 0.45] \times [0.25, 0.35] \times [0.35, 0.45]$	$\pm 10^{-2}$
5	3	50	0.2	6	156	$[0.3, 0.4] \times [0.3, 0.4] \times [-0.4, -0.3]$	$\pm 10^{-2}$
6	3	50	0.2	5	106	$[0.35, 0.4] \times [-0.35, -0.3] \times [0.35, 0.4]$	$\pm 10^{-3}$
7	3	20	0.5	2	500	$[0.35, 0.45] \times [0.45, 0.55] \times [0.25, 0.35]$	$\pm 10^{-2}$
8	4	25	0.2	5	106	$[0.5, 0.6]^4$	$\pm 10^{-4}$
9	4	20	1	3	300	$[0.6, 0.7] \times [-0.7, -0.6] \times [-0.4, -0.3] \times [0.5, 0.6]$	$\pm 10^{-3}$
10	4	50	0.2	1	500	$[9.5, 9.55] \times [-4.5, -4.45] \times [2.1, 2.11] \times [1.5, 1.51]$	$\pm 10^{-4}$

in Flow*, and the queue size is set to be 200. As an example, we illustrate the flowpipes computed for the benchmark #9 in Figure 5.

Car example We trained a neural network controller, for the unicylce model of a car as a stabilizing controller. An MPC controller was used to train the network, which ends up having interesting dynamics. We were able to compute the reach sets for this case, which are shown in Fig 6.

Quadrotor example. We start with the initial set which is a box with the maximum width 0.01, and try to compute the flowpipes for the time horizon [0, 10]. We use a TM order 5 with the integration stepsize 0.01, the maximum error encountered in the PWL approximations is bounded by 1.8e−4.

8 CURRENT LIMITATIONS

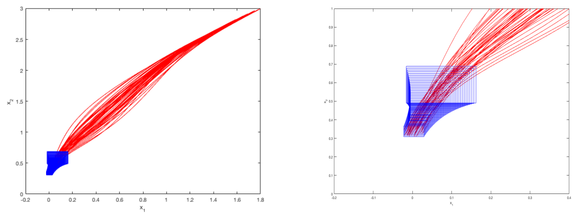


Figure 7: Flowpipes for the Drone Model

Our approach also provides a way to alleviate the wrapping effect in reachability analysis for neural feedback systems by approximating neural networks locally as polynomials plus intervals. However, it may lead to difficulties that arise primarily due to the following limitations:

- *Large initial sets.* Large initial sets either cause large errors for the local approximation or require high degree polynomials for approximations.
- *Divergent traces.* Traces of dynamical systems can diverge (eg., positive Lyapunov exponent) locally before converging, as see in Fig. 7. In such cases, our method may not control the explosion of overestimation. Currently, such cases can be handled through a subdivision of the state-space which can be expensive for a large model.

Solving these two problems will continue to drive our future efforts in this space.

9 CONCLUSION

Thus, we have presented an approach to compute accurate flowpipe overapproximations for the reachable sets of neural feedback systems. Our key contribution is a sound rule generation method along with a rigorous error analysis technique, based on which the wrapping effect in flowpipe computation is greatly reduced. Future directions will investigate stochastic uncertainties in our framework.

Acknowledgments: This work was supported in part by the Air Force Research Laboratory (AFRL) and by the US NSF under Award # 1646556.

REFERENCES

[1] Abadi, Martín et al. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proc. OSDI'16*. USENIX, 265–283.

[2] M. Althoff. 2015. An Introduction to CORA 2015. In *Proc. of ARCH'15 (EPiC Series in Computer Science)*, Vol. 34. EasyChair, 120–151.

[3] S. Bak and M. Caccamo. 2013. Computing Reachability for Nonlinear Systems with HyCreate. In *Demo and Poster Session in HSCC'13*.

[4] S. Bak and P. S. Duggirala. 2017. HyLAA: A Tool for Computing Simulation-Equivalent Reachability for Linear Systems. In *Proc. of HSCC'17*. ACM, 173–178.

[5] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. 2016. Measuring neural net robustness with constraints. In *Advances in Neural Information Processing Systems*. 2613–2621.

[6] M. Berz. 1999. *Modern Map Methods in Particle Beam Physics*. Advances in Imaging and Electron Physics, Vol. 108. Academic Press.

[7] M. Berz and K. Makino. 1998. Verified Integration of ODEs and Flows Using Differential Algebraic Methods on High-Order Taylor Models. *Reliable Computing* 4 (1998), 361–369. Issue 4.

[8] X. Chen. 2015. *Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models*. Ph.D. Dissertation. RWTH Aachen University.

[9] X. Chen, E. Ábrahám, and S. Sankaranarayanan. 2012. Taylor Model Flowpipe Construction for Non-linear Hybrid Systems. In *Proc. of RTSS'12*. IEEE Computer Society, 183–192.

[10] X. Chen, E. Ábrahám, and S. Sankaranarayanan. 2013. Flow*: An Analyzer for Non-linear Hybrid Systems. In *Proc. of CAV'13 (LNCS)*, Vol. 8044. Springer, 258–263.

[11] X. Chen and S. Sankaranarayanan. 2016. Decomposed Reachability Analysis for Nonlinear Systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*. IEEE Press, 13–24.

[12] Antonio Eduardo Carrilho da Cunha. 2015. Benchmark: Quadrotor Attitude Control. In *Proc. of ARCH 2015 (EPiC Series in Computing)*, Vol. 34. EasyChair, 57–72.

Table 2: Details of the experiments for different Benchmarks. Legend : #: Benchmark No., k : TM Integration Order , τ_I : stepsize used for flowpipe construction, P_o :, order of the polynomial used for the regression, ϵ : maximum computed error bound between the neural network and polynomial, T_p: time cost for computing the reach sets using polynomial rule generation, T_I: time taken for computing the reachable sets using simple interval propagation, P_r : % of the time cost in polynomial regression, P_{pwl} : % of the time cost in computing the Piecewise Linear Approximations for the polynomials generated , P_s : % of the time cost in Sherlock for computing the error, P_f : % of the time cost in Flow* to compute the reachable sets for the ODE. T_I: time cost of a direct combination of Flow* and Sherlock, L_c : Maximum number of linear regions in one control step .

#	k	τ_I	P_o	ϵ	T_p (s)	$P_r(\%)$	$P_{pwl}(\%)$	$P_s(\%)$	$P_f(\%)$	T_I (s)	L_c
1	4	0.02	2	0.66	6.5	2.2	2.3	14	81	×	31
2	5	0.02	2	0.2	46.0	1.3	1.4	42	54	×	31
3	4	0.02	2	1.89e-2	40.4	1.3	0.9	11	86	×	7
4	5	0.02	2	3.7e-2	21.8	2.4	4.2	62.6	30.2	×	76
5	4	0.02	2	6.8e-5	19.5	2.7	1.2	44.7	50.6	×	4
6	4	0.02	2	2.7e-2	15.3	2.3	1.7	12.0	82.7	×	6
7	5	0.05	2	1.2e-2	57.4	1.9	0.3	93	5	×	58
8	4	0.02	2	6e-2	13.1	1.87	7	13.3	75.3	×	156
9	4	0.1	2	6.8e-2	36.7	1.5	2.0	80	16.1	×	86
10	30	0.01	2	0.02	1081	0.4	0.1	0.85	98.3	×	16

[13] Luiz H. de Figueiredo and Jorge Stolfi. 1997. Self-Validated Numerical Methods and Applications. In *Brazilian Mathematics Colloquium monograph.* IMPA, Rio de Janeiro, Brazil. Cf. http://www.ic.unicamp.br/~stolfi/EXPORT/papers/by-tag/fig-sto-97-iaaa.ps.gz.

[14] P. S. Duggirala, S. Mitra, M. Viswanathan, and M. Potok. 2015. C2E2: A Verification Tool for Stateflow Models. In *Proc. of TACAS'15 (LNCS)*, Vol. 9035. Springer, 68–82.

[15] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. 2018. Learning and verification of feedback control systems using feedforward neural networks. *IFAC-PapersOnLine* 51, 16 (2018), 151–156.

[16] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. 2018. Output Range Analysis for Deep Feedforward Neural Networks. In *NASA Formal Methods*, Aaron Dutle, César Muñoz, and Anthony Narkawicz (Eds.). Springer International Publishing, Cham, 121–138.

[17] Rüdiger Ehlers. 2017. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *ATVA (Lecture Notes in Computer Science)*, Vol. 10482. Springer, 269–286.

[18] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. 2011. SpaceEx: Scalable Verification of Hybrid Systems. In *Proc. of CAV'11 (LNCS)*, Vol. 6806. Springer, 379–395.

[19] LiMin Fu. 1994. Rule generation from neural networks. *IEEE Transactions on Systems, Man, and Cybernetics* 24, 8 (Aug 1994), 1114–1124.

[20] E. Hainry. 2008. Reachability in Linear Dynamical Systems. In *Proc. of CiE 2008 (LNCS)*, Vol. 5028. Springer, 241–250.

[21] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2016. Safety Verification of Deep Neural Networks. *CoRR* abs/1610.06940 (2016). http://arxiv.org/abs/1610.06940

[22] M. Jankovic, D. Fontaine, and P. V. Kokotovic. 1996. TORA example: cascade- and passivity-based control designs. *IEEE Transactions on Control Systems Technology* 4, 3 (1996), 292–297.

[23] Kyle Julian and Mykel J. Kochenderfer. 2017. Neural Network Guidance for UAVs. In *AIAA Guidance Navigation and Control Conference (GNC).* https://doi.org/10.2514/6.2017-1743

[24] R. R. Kadiyala. 1993. A tool box for approximate linearization of nonlinear systems. *IEEE Control Systems* 13, 2 (April 1993), 47–57. https://doi.org/10.1109/37.206985

[25] Gregory Kahn, Tianhao Zhang, Sergey Levine, and Pieter Abbeel. 2016. PLATO: Policy Learning using Adaptive Trajectory Optimization. *arXiv preprint arXiv:1603.00622* (2016).

[26] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. *Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks.* Springer International Publishing, Cham, 97–117. https://doi.org/10.1007/978-3-319-63387-9_5

[27] S. Kong, S. Gao, W. Chen, and E. M. Clarke. 2015. dReach: δ-Reachability Analysis for Hybrid Systems. In *Proc. of TACAS'15 (LNCS)*, Vol. 9035. Springer, 200–205.

[28] Lectures. 2014. Nonlinear Systems and Control. http://people.ee.ethz.ch/~apnoco/Lectures2014/.

[29] Alessio Lomuscio and Lalit Maganti. 2017. An approach to reachability analysis for feed-forward ReLU neural networks. *CoRR* abs/1706.07351 (2017). arXiv:1706.07351 http://arxiv.org/abs/1706.07351

[30] K. Makino and M. Berz. 2003. Taylor models and other validated functional inclusion methods. *J. Pure and Applied Mathematics* 4, 4 (2003), 379–456.

[31] S. Mitra and Y. Hayashi. 2000. Neuro-fuzzy rule generation: survey in soft computing framework. *IEEE Transactions on Neural Networks* 11, 3 (May 2000), 748–768.

[32] R. E. Moore, R. B. Kearfott, and M. J. Cloud. 2009. *Introduction to Interval Analysis.* SIAM.

[33] A. Neumaier. 1993. *The Wrapping Effect, Ellipsoid Arithmetic, Stability and Confidence Regions.* Springer Vienna, 175–190.

[34] W. Perruquetti, J. P. Richard, and P. Borne. 1996. Lyapunov analysis of sliding motions: Application to bounded control. *Mathematical Problems in Engineering* 3, 1 (1996), 1 – 25.

[35] Luca Pulina and Armando Tacchella. 2010. An abstraction-refinement approach to verification of artificial neural networks. In *Computer Aided Verification.* Springer, 243–257.

[36] Luca Pulina and Armando Tacchella. 2012. Challenging SMT Solvers to Verify Neural Networks. *AI Commun.* 25, 2 (2012), 117–135.

[37] Nathalie Revol and Fabrice Rouillier. 2005. Motivations for an Arbitrary Precision Interval Arithmetic and the MPFI Library. *Reliable Computing* 11, 4 (2005), 275–290. https://doi.org/10.1007/s11155-005-6891-y

[38] Kazumi Saito and Ryohei Nakano. 2002. Extracting regression rules from neural networks. *Neural Networks* 15, 10 (2002), 1279 – 1288.

[39] Hanan J. Samet. 2006. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann.

[40] Mohamed Amin Ben Sassi, Ezio Bartocci, and Sriram Sankaranarayanan. 2017. A Linear Programming-based iterative approach to Stabilizing Polynomial Dynamics. In *Proc. IFAC'17.* Elsevier.

[41] Karsten Scheibler, Leonore Winterer, Ralf Wimmer, and Bernd Becker. 2015. Towards verification of artificial neural networks. In *MBMV Workshop.* 30âĂŞ40.

[42] Richard S. Sutton and Andrew G. Barto. 2017. *Reinforcement Learning: An Introduction.* MIT Press.

[43] H. Tsukimoto. 2000. Extracting rules from trained neural networks. *IEEE Transactions on Neural Networks* 11, 2 (Mar 2000), 377–389.

[44] Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. 2017. Output Reachable Set Estimation and Verification for Multi-Layer Neural Networks. *CoRR* abs/1708.03322 (2017). arXiv:1708.03322 http://arxiv.org/abs/1708.03322

[45] Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. 2107. Reachable Set Computation and Safety Verification for Neural Networks with ReLU Activations. Cf. https://arxiv.org/pdf/1712.08163.pdf, posted on ArXIV Dec. 2017.

[46] Weiming Xiang, Hoang-Dung Tran, Joel A. Rosenfeld, and Taylor T. Johnson. 2018. Reachable Set Estimation and Verification for a Class of Piecewise Linear Systems with Neural Network Controllers. To Appear in the American Control Conference (ACC), invited session on Formal Methods in Controller Synthesis.

[47] Dong-Hae Yeom and Young Hoon Joo. 2012. Control Lyapunov Function Design by Cancelling Input Singularity. 12 (06 2012).

A APPENDIX: PROOFS OF THEOREMS

We will now discuss the proofs of the various theorems stated in the paper. We first consider Theorem 5.2 from page 6.

Theorem A.1. *For any polynomial f, sample $\mathbf{x}$, set S, tolerance ϵ and minimum box width δ, the FindMaxInterval routine (a) always terminates; (b) if it succeeds, returns a box B such that $f(B) \subseteq [-\epsilon, \epsilon]$.*

Proof. To prove that the FindMaxInterval routine (Algorithm 2) always terminates, we need to prove that at each iteration of the while loop (line 9) at least one of two progress conditions are met: (a) the size of `wlist` decreases (else branch at line 17 or else branch at line 14, or (b) the size of the list remains the same, but the volume of the interval $[\mathbf{a}, \mathbf{b}]$ strictly increases by at least δ^n (then branches taken at lines 14 and 17). Consider a lexicographic ranking function $(|\text{wlist}|, -\Pi_{j=1}^{n}(\mathbf{b}_j - \mathbf{a}_j))$. We note that $|\text{wlist}| \geq 0$ and the volume of $[\mathbf{a}, \mathbf{b}]$ is upper bounded by that of S.

To establish (b), we will prove the loop invariant that **EvaluateRange**$(f, [\mathbf{a}, \mathbf{b}]) \subseteq [-\epsilon, \epsilon]$. This clearly holds the first time the head of the while loop is visited (line 9) and each time $[\mathbf{a}, \mathbf{b}]$ is updated in a loop iteration, the loop invariant is re-established (line 17). The rest follows by assuming the soundness property of the **EvaluateRange** routine. □

Next, we consider the proof of theorem 5.4 in page 7. Let $D^2 f(\mathbf{x})$ represent the Hessian matrix of a C^2 function $f : \mathbb{R}^n \mapsto \mathbb{R}$. The i, j entry of the Hessian is $\frac{\partial^2 f}{\partial x_i \partial x_j}$. For a $n \times n$ symmetric matrix M, let $\lambda_{\max}(M)$ be the largest eigen value of M and $\lambda_{\min}(M)$ be the smallest eigenvalue. These are always real numbers and well-known to be a continuous function of the matrix M. Finally, recall that for a quadratic form $q : \mathbf{x}^t A \mathbf{x}$, we have the inequality that $\lambda_{\min}(A)\mathbf{x}^t\mathbf{x} \leq \mathbf{x}^t A \mathbf{x} \leq \lambda_{\max}(A)\mathbf{x}^t\mathbf{x}$. The Euclidean norm of a vector $||\mathbf{x}||_2$ is simply $\mathbf{x}^t\mathbf{x}$.

Theorem A.2. *For any compact set D, and fixed tolerance $\epsilon > 0$, there exists a sound procedure **EvaluateRange** and a corresponding value of δ such that the assertion check in line 5 of algorithm 2 always succeeds.*

Proof. Using Lemma 5.3, we know that the function f and sample $\mathbf{x}$ satisfy $f(\mathbf{x}) = 0$ and $\nabla f(\mathbf{x}) = 0$. Define

$$N(\mathbf{x}) : \lambda_{\min}(D^2 f(\mathbf{x})), \text{ and } M(\mathbf{x}) : \lambda_{\max}(D^2 f(\mathbf{x})),$$

the smallest and largest eigenvalues of the Hessian matrix of f evaluated at $\mathbf{x}$. Note that M is a scalar function of $\mathbf{x}$ and is continuous.

Now let us choose some δ. Using a Taylor series development of f, we note

$$f(\mathbf{x} + \mathbf{h}) : \underbrace{f(\mathbf{x}) + \nabla f(\mathbf{x}) \cdot \mathbf{h}}_{=0} + \frac{1}{2}\mathbf{h}^t D^2 f(\hat{\mathbf{x}})\mathbf{h},$$

for some $\hat{\mathbf{x}} : \mathbf{x} + \alpha\mathbf{h}$. The first two terms vanish due to Lemma 5.3.

Therefore, the **EvaluateRange**$(f, [\mathbf{a}, \mathbf{b}])$ procedure is simply as follows: (a) set $m_0 : \max_{\mathbf{z} \in [\mathbf{a}, \mathbf{b}]} M(\mathbf{z})$ and $n_0 : \min_{\mathbf{z} \in [\mathbf{a}, \mathbf{b}]} N(\mathbf{z})$. (b) $\beta : \frac{1}{2}\max(|m_0|, |n_0|)n||\mathbf{b} - \mathbf{a}||^2$, and (c) return **EvaluateRange**$(f, [\mathbf{a}, \mathbf{b}]) : [-\beta, \beta]$.

The soundness of the procedure follows from Taylor theorem. Note that if $\mathbf{x} + \mathbf{h} \in [\mathbf{a}, \mathbf{b}]$ then $||\mathbf{h}||_2^2 \leq ||\mathbf{b} - \mathbf{a}||^2$. Therefore,

$$|f(\mathbf{x} + \mathbf{h})| = |\frac{1}{2}\mathbf{h}^t D^2 f(\hat{\mathbf{x}})\mathbf{h}|$$

We know that $\mathbf{h}^t D^2 f(\hat{\mathbf{x}})\mathbf{h} \leq M(\mathbf{x})\mathbf{h}^t\mathbf{h} \leq m_0||\mathbf{h}||_2^2$. Furthermore, $\mathbf{h}^t D^2 f(\hat{\mathbf{x}})\mathbf{h} \geq N(\mathbf{x})\mathbf{h}^t\mathbf{h} \geq n_0||\mathbf{h}||_2^2$. Therefore, $|\mathbf{h}^t D^2 f(\hat{\mathbf{x}})\mathbf{h}| \leq \max(|m_0|||\mathbf{h}||_2^2, |n_0|||\mathbf{h}||_2^2) \leq \max(|m_0|, |n_0|)||\mathbf{h}_2^2||$. Putting it all together, we have

$$|f(\mathbf{x} + \mathbf{h})| = |\frac{1}{2}\mathbf{h}^t D^2 f(\hat{\mathbf{x}})\mathbf{h}| \leq \beta.$$

Next, given ϵ, we choose δ as follows.

Since D is compact and $S \subseteq D$. Therefore, let us define m^* as

$$m^* : \max(|\max_{\mathbf{x} \in D} M(\mathbf{x})|, |\min_{\mathbf{x} \in D} N(\mathbf{x})|).$$

The compactness of D and continuity of $M(\mathbf{x}), N(\mathbf{x})$ guarantee that m^* exists. If $m^* = 0$, then the second derivative vanishes everywhere and f is essentially the 0 function. For such a function, the assertion in line 5 will never fail. Without loss of generality, let $m^* > 0$.

Consider the box $B_0 : [\mathbf{x} - \frac{\delta}{2}\mathbf{1}, \mathbf{x} + \frac{\delta}{2}\mathbf{1}]$, chosen in line 2 of Algorithm 2. Let us set $\frac{1}{2}m^* n\delta^2 = \epsilon$, or in other words, $\delta : \sqrt{\frac{2\epsilon}{m^* n}}$. We note that for any $\mathbf{x}+\mathbf{h} \in B_0$, $|f(\mathbf{x}+\mathbf{h})| \leq \frac{1}{2}m_0||\mathbf{h}||_2^2 \leq \frac{1}{2}m^* n\delta^2 \leq \epsilon$. Thus the assertion in line 5 will never fail if the value of δ is set to at most $\sqrt{\frac{2\epsilon}{m^* n}}$ and the **EvaluateRange** function defined in this proof is used. □

Next, we will consider the proof of theorem 5.5 from page 7.

Theorem A.3. *If Algorithm 1 terminates with success for input p over domain D with tolerance ϵ, then the resulting set of linear pieces L define a PWL function f_L such that $|f_L(\mathbf{x}) - p(\mathbf{x})| \leq \epsilon$ for each $\mathbf{x} \in D$.*

Proof. We conclude that the final result f_L must be a function defined over the domain D. This is proved using a loop invariant that $\bigcup_{\langle B_s, \mathbf{c}, d\rangle \in L} B_s \cup S = D$ holds for the while loop in line 4. Another useful loop invariant to prove is that for all pieces $(B, \mathbf{c}, d) \in L$, we have $B \cap S = \emptyset$ at the loop head (line 4). Next, we note that the domain of the pieces are mutually exclusive. This is proved by noting that the set B_s returned at line 8 must satisfy $B_s \subseteq S$. Therefore, B_s cannot have a common intersection with any existing piece in L. Together, we note that the linearization defined by f_L exists and Note that line 9 in Algorithm 1 follows directly from Theorem 5.2. Therefore, the property $|f_L(\mathbf{x}) - p(\mathbf{x})| \leq \epsilon$ holds for each piece added to L in line 11. □

Finally, we will address the proof of Theorem 6.3 in page 8.

Theorem A.4. *For all $\mathbf{x} \in D$, If $z = f_L(\mathbf{x})$ then $(\exists\, \vec{l} \in \{0, 1\}^N)\ \Psi_L(\mathbf{x}, z, \vec{l})$.*

Proof. Suppose for some $\mathbf{x} \in D$, we have that $z = f_L(\mathbf{x})$. Then, $\mathbf{x}$ must belong to precisely one linear piece in $\mathbf{x}$. Therefore, let it belong to piece corresponding to $\vec{l}_j$. We will set $\vec{l}_j = 1$ and $\vec{l}_i = 0$ for all $i \neq j$. We now verify that (6.1), (6.2) and (6.3) are all satisfied by the assignment to $\vec{l}$. □

Table 3: ODE for the different Benchmarks.

#	Benchmark ODE
1	$\dot{x_1} = x_2 - x_1^3 + w, \dot{x_2} = u$
2	$\dot{x_1} = x_2, \dot{x_2} = ux_2^2 - x_1 + w$
3	$\dot{x_1} = -x_1(0.1+(x_1+x_2)^2), \dot{x_2} = (u+x_1+w)(0.1+(x_1+x_2)^2)$
4	$\dot{x_1} = x_2 + 0.5x_3^2, \dot{x_2} = x_3 + w, \dot{x_3} = u$
5	$\dot{x_1} = -x_1+x_2-x_3+w, \dot{x_2} = -x_1(x_3+1)-x_2, \dot{x_3} = -x_1 + u$
6	$\dot{x_1} = -x_1^3 + x_2, \dot{x_2} = x_2^3 + x_3, \dot{x_3} = u + w$
7	$\dot{x_1} = x_3^3 - x_2 + w, \dot{x_2} = x_3, \dot{x_3} = u$
8	$\dot{x_1} = x_2, \dot{x_2} = -9.8x_3 + 1.6x_3^3 + x_1x_4^2, \dot{x_3} = x_4, \dot{x_4} = u$
9	$\dot{x_1} = x_2, \dot{x_2} = -x_1 + 0.1sin(x_3), \dot{x_3} = x_4, \dot{x_4} = u$
10	$\dot{x_1} = x_4cos(x_3), \dot{x_2} = x_4sin(x_3), \dot{x_3} = u_2, \dot{x_4} = u_1 + w$

B APPENDIX: DETAILS OF BENCHMARKS AND EXPERIMENTAL RESULTS

We give details of the benchmarks in Table 1 and present the plots of the flowpipes.

High Dimensional Example

We refer the reader to [12], for further details on the system dynamics. The initial set is given by the following : $p_n \in [-1, -0.99]$, $p_e \in [-1, -0.99]$, $h \in [9, 9.01]$, $u \in [-1, -0.99]$, $v \in [-1, -0.99]$, $w \in [-1, -0.99]$, $q_0 \in [0, 0]$, $q_1 \in [0, 0]$, $q_2 \in [0, 0]$, $q_3 \in [1, 1]$, $p \in [-1, -0.99]$, $q \in [-1, -0.99]$, $r \in [-1, -0.99]$, $p_I \in [0, 0]$, $q_I \in [0, 0]$, $r_I \in [0, 0]$, $h_I \in [0, 0]$. The ODE equations governing the dynamics are given by the following, where d is the time-varying disturbance.

$$
\begin{aligned}
\dot{p_n} &= 2u(q_0^2 + q_1^2 - 0.5) - 2v(q_0q_3 - q_1q_2) + 2w(q_0q_2 + q_1q_3)\\
\dot{p_e} &= 2v(q_0^2 + q_2^2 - 0.5) + 2u(q_0q_3 + q_1q_2) - 2w(q_0q_1 - q_2q_3)\\
\dot{h} &= 2w(q_0^2 + q_3^2 - 0.5) - 2u(q_0q_2 - q_1q_3) + 2v(q_0q_1 + q_2q_3)\\
\dot{u} &= rv - qw - 11.62(q_0q_2 - q_1q_3)\\
\dot{v} &= pw - ru + 11.62(q_0q_1 + q_2q_3)\\
\dot{w} &= qu - pv + 11.62(q_0^2 + q_3^2 - 0.5) + \text{control\_input} + d\\
\dot{q_0} &= -0.5q_1p - 0.5q_2q - 0.5q_3r\\
\dot{q_1} &= 0.5q_0p - 0.5q_3q + 0.5q_2r\\
\dot{q_2} &= 0.5q_3p + 0.5q_0q - 0.5q_1r\\
\dot{q_3} &= 0.5q_1q - 0.5q_2p + 0.5q_0r\\
\dot{p} &= (-40.000632584p_I - 2.8283979829540p) - 1.133407423682qr\\
\dot{q} &= (-39.999804525q_I - 2.8283752541008q) + 1.132078179614pr\\
\dot{r} &= (-39.999789097r_I - 2.8284134223281r) - 0.004695219978pq\\
\dot{p_I} &= p, \dot{q_I} = q, \dot{r_I} = r, \dot{h_I} = h
\end{aligned}
$$

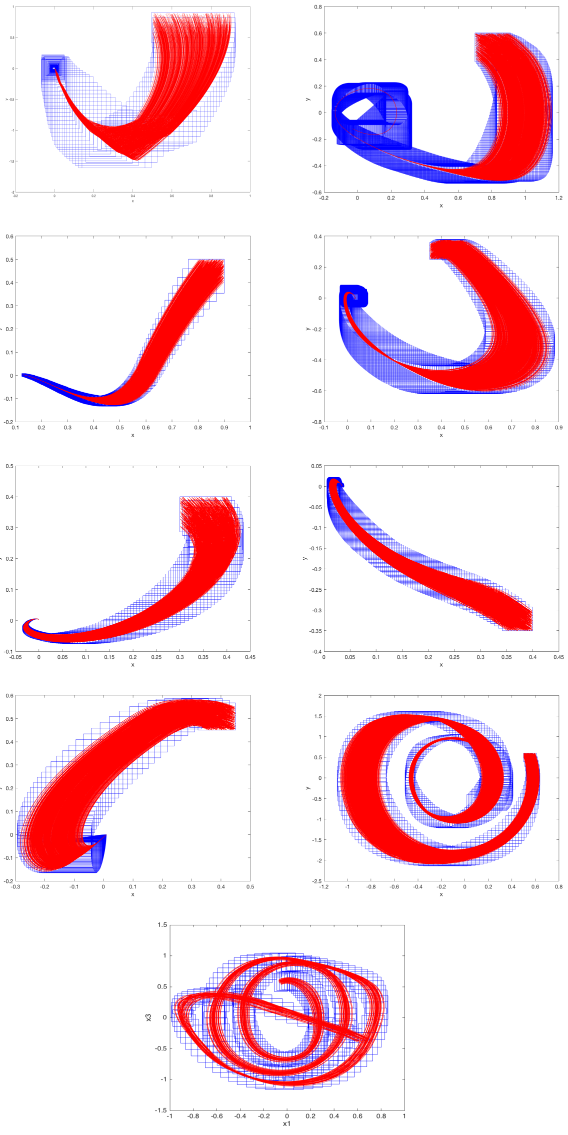


Figure 8: Flowpipes computed for different benchmarks 1 - 9 (left to right and top down). The red trajectories are the simulation traces.

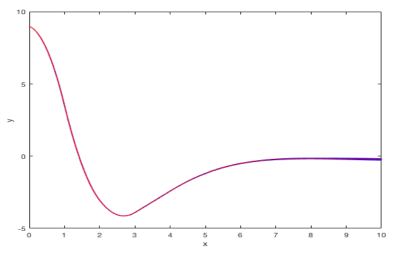


Figure 9: Flowpipes computed for the quadrotor model. The red trajectories are the simulation traces.