NExG: Provable and Guided State Space Exploration of Neural Network Control Systems using Sensitivity Approximation

Manish Goyal, Miheer Dewaskar, Parasara Sridhar Duggirala

***Abstract*—We propose a new technique for performing state space exploration of closed loop control systems with neural network feedback controllers. Our approach involves approximating the sensitivity of the trajectories of the closed loop dynamics. Using such an approximator and the system simulator, we present a guided state space exploration method that can generate trajectories visiting the neighborhood of a target state at a specified time. We present a theoretical framework which establishes that our method will produce a sequence of trajectories that will reach a suitable neighborhood of the target state. We provide thorough evaluation of our approach on various systems with neural network feedback controllers of different configurations. We outperform earlier state space exploration techniques and achieve significant improvement in both the quality (explainability) and performance (convergence rate). Finally, we adopt our algorithm for the falsification of a class of temporal logic specification, assess its performance, and show its potential in supplementing existing falsification algorithms.**

***Index Terms*—Closed loop control systems, neural networks, sensitivity function, state space exploration, falsification.**

I. Introduction

DESIGN and verification of closed loop systems has become an increasingly challenging task. First, advances in hardware and software have made it easier to integrate sophisticated control algorithms in embedded systems. Second, control designers now often integrate multiple technologies and satisfy ever increasing behavioral specifications expected from complex Cyber-physical systems (CPS). Third, the non-linearity in the behaviors of the closed loop systems makes it difficult to predict the outcomes of perturbations in the state or the environment. Control design for linear systems typically involves techniques such as pole placement and computing Lyapunov functions. However, such analytical method usually do not scale well to hybrid or non-linear systems encountered in real world applications. Therefore, we have witnessed a surge in neural network based control design in recent times [38], [53]. But despite its desired utility, neural network controller, due to its characteristics and behavior, adds to the complexity of the underlying system thus making it more difficult to perform safety analysis.

M. Goyal and P. S. Duggirala are with the Department of Computer Science, University of North Carolina at Chapel Hill, NC. M. Dewaskar is with the Department of Statistical Science, Duke University, NC. e-mails: {manishg,psd}@cs.unc.edu, miheer.dewaskar@duke.edu.

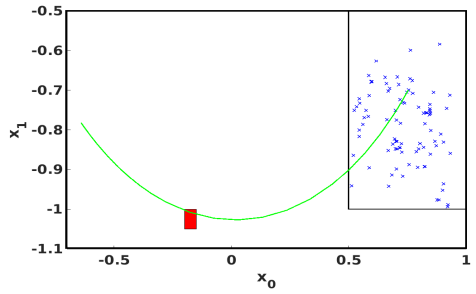
Manuscript received April 07, 2022; revised June 11, 2022; accepted July 05, 2022. This article was presented at the International Conference on Embedded Software (EMSOFT) 2022 and appeared as part of the ESWEEK-TCAD special issue.

In a typical work flow, the control designer designs a control algorithm and generates a few test cases to check if the specification is satisfied or violated. However, because of the increasing system complexity, these test cases often do not generalize to the system behavior at large. This is especially difficult if one has to consider all possible inter leavings of the continuous and discrete behaviors encountered by a modern CPS. Due to different sequence of mode changes, two neighboring states can potentially have divergent trajectories, thus extrapolating the behavior from one state to another becomes challenging. The problem is further exacerbated by sophisticated neural network based control algorithms. Since such neural network controllers are typically learned from a finite number of samples, a designer needs to perform additional checks for controller's behavior outside the test suite. However, such manual validation is not practically feasible.

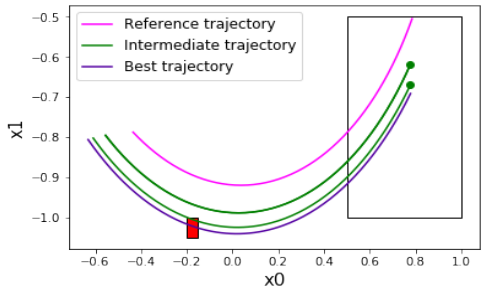
In some instances, the specification is mathematically expressed as as a temporal logic formula that is used by an off-the-shelf falsification tool for automatically generating a trajectory that violates the specification. But such an approach has a few drawbacks. Falsification tools are primarily geared towards finding a violating trace for the given specification, not necessarily to help the designer in systematically exploring the state space. Moreover, the search for a counterexample is performed using stochastic optimization and gradient descent methods. The optimization engine generates random trajectories which would not yield much intuition for the designer about two neighboring behaviors. Second, if the control designer changes the specification during testing, the results from the previous runs may no longer be useful. Third, existing falsification tools require the specification to be provided in a temporal logic such as signal temporal logic or metric temporal logic (STL/MTL). The designer needs to understand these specification languages which despite being useful in the verification phase, may cause hindrance during the design and exploration phases.

State space exploration entails systematically generating trajectories to explore desired (or undesired) outcomes of the system. For example, for a given safety specification, a designer might like to generate test cases that are close to satisfying or violating the specification. Existing falsification tools are neither capable of obtaining such executions nor informing the control designer about the additional tests to conduct for validating safety, measuring coverage or exploring new regions. Although NeuralExplorer [29] alleviates some of these concerns by enabling the control designer to perform

systematic state space exploration, it suffers from high training time, and it lacks convergence to the true solution as well as a theoretical analysis.



(a) from S-TaLiRo



(b) from NExG

Figure 1. Illustration of Falsification schemes. The red-colored box is the unsafe set U and the inner while-colored box is the initial set. While S-TaLiRo conducts falsification in a stochastic manner and takes significant number of iterations to find a falsifying execution to the given safety specification, NExG finds a valid counterexample in a fewer iterations, that too, in a much more directed manner.

In this paper, we present a technique that uses neural network approximations of sensitivity over small ranges to to perform state space exploration of closed loop systems with neural network feedback controllers. The *sensitivity* of a closed loop system at an initial state measures the change in the system trajectory as a result of perturbing the initial state. Sensitivity can help the designer in developing an intuition about the convergence and divergence of system behaviors. Consequently, the designer can take an active role in systematically navigating the space of new test cases during control design. Thus instead of generating executions using a stochastic optimization solver for falsification, our approach explores the state space in a more systematic manner. As a result, it takes less number of iterations to generate desirable executions as illustrated in Figure I.

Since our framework only only requires system traces, it is generalizable to black-box systems in the absence of precise analytical models. Similar to other simulation driven analyses, our technique also depends on using system simulations to tap key information or properties about the system. Further, the motivation behind employing neural networks is not only driven by their power to approximate complex functions but also because hardware and software advancements have made these neural networks easy to train and deploy. We believe that such automated state space exploration is not only useful in man-machine collaborative test case generation, but also for designing safe neural network feedback functions for closed loop control systems.

This paper makes multiple contributions. (i) It presents a new state space exploration algorithm NExG, which is an extension of NeuralExplorer [29] for inverse sensitivity learned over small perturbations. (ii) It provides theoretical guarantees supported by empirical results. The paper demonstrates that NExG will converge to a neighborhood of the target point even if the learned neural network only approximates the inverse sensitivity function, and it performs much better than NeuralExplorer while requiring less computational resources. (iii) It performs extensive evaluation on 20 standard non-linear benchmarks with up-to 6 dimensions state spaces having neural network feedback controller with multiple layers. (iv) It presents a simple inverse sensitivity based falsification algorithm for a class of temporal logic safety specifications. The evaluations exhibit that the presented falsification scheme is capable of finding a more robust falsifying trajectory in significantly less number of iterations as compared to a widely used falsification tool S-TaLiRo [4]. (v) Finally, it presents additional features of the framework such as computing set coverage and customized state space exploration.

II. Related Work

Verification or *Reachability analysis* is typically aimed at verifying safety specification(s) of the safety critical control system [3], [11]. Some of the notable works in this domain are SpaceEx [23], Flow* [8], CORA [2] and HyLAA [5]. These tools use different symbolic representations such as support functions, generalized star etc. for the set of reachable states. While these techniques are useful for proving that the safety specification is satisfied, some other recent works [25], [26], [28] have explored using reachability analysis to generate counterexamples of interest.

Falsification is employed to generate executions that violate a given safety specification [15], [20] instead of proving safety. In these techniques, the required specification is expressed as a formula in temporal logic such as Metric Temporal Logic (MTL) [36] or Signal Temporal Logic (STL) [37], [42]. For a given temporal logic specification, falsification techniques use various heuristics [1], [12], [24], [46], [49], [57] in an attempt to generate trajectories that violate the specification. Two well known falsification tools are S-TaLiRo [4] and Breach [13]. Another work [6] uses symbolic reachability supplemented by trajectory splicing to scale up hybrid system falsification.

Simulation based state space exploration [10], [14] and verification [16], [21], [32] have also shown some promise by taking the advantages of symbolic and analytical techniques. Such methods either use bounds on sensitivity [21], [32] to obtain an overapproximation of the reachable set or require analytical model to perform random exploration of the state space [10]. While these techniques can bridge the gap between falsification and verification, they might still suffer due to high system dimensionality and complexity. That is, the number of required trajectories may increase exponentially with system dynamics and dimensions. C2E2 [17], and DryVR [22] are some of the well known tools in this domain.

Given the rich history of application of neural networks in control [39], [44], [45] and the recent advances in software and hardware platforms, neural networks are now being deployed in various control tasks. Consequently, many verification techniques are being developed for neural network based control systems [18], [33], [51], [54], [56] and some other domains [31], [52]. Many neural network based frameworks for learning the dynamics or their properties have been also proposed in recent times [7], [40], [48], which further underlines the need of an efficient state space exploration.

In the model checking domain, neural networks have been used for state classification [47] as well as reachability analysis by learning state density distribution [43] or reachability function in NeuReach [50]. In contrast, NExG learns sensitivity functions and is geared towards state space exploration. While NeuralExplorer [29] also learns the sensitivity functions, NExG approximates the sensitivity of closed loop control systems for small perturbations. The corresponding change in the neural network training framework reduces its training time by considerable amount which is further supplemented with the choice of a uniform network architecture unlike [29]. Two parameters, *scaling factor* and *correction period*, are introduced in NExG to maintain the trade off between approximation error and the number of simulations generated. Another distinguishing feature is that we provide theoretical guarantees for NExG for its convergence.

III. Preliminaries

The state of the system is an element typically denoted as $x \doteq (\mathbf{x}_1, \ldots, \mathbf{x}_n) \in \mathbb{R}^n$. For $v \in \mathbb{R}^n$, let $\|v\|$ denote the standard Euclidean norm of the vector v. For $\delta \geq 0$, $B_\delta(x) \doteq \{x' \in \mathbb{R}^n \mid \|x - x'\| \leq \delta\}$ is the closed neighborhood around x of radius δ. We will denote the state space of the system by $\mathbb{D} \subseteq \mathbb{R}^n$, and the dynamics of the plant as

$$\dot{x} = f(x, u)$$

where $x \in \mathbb{D}$ is the state of the system and $u \in \mathbb{R}^m$ is the input. A closed loop system is a control system where the process or the system is regulated by a feedback control action which is automatically computed as a function of system output. Suppose we use a feedback-function (also called a controller) g that is regulated by the system output, i.e. $u = g(x)$, then have a *closed loop* system that satisfies

$$\dot{x} = f\left(x, g(x)\right). \tag{1}$$

We will assume that f and g are such that (1) has a unique solution $x : \mathbb{R} \to \mathbb{D}$ satisfying $x(0) = x_0$ for every $x_0 \in \mathbb{D}$. For example, by existence and uniqueness theorem for differential equations, this condition is guaranteed if $\mathbb{D} = \mathbb{R}^n$ and both f and g are Lipschitz functions of their inputs.

Definition 1. *Let $\xi(x_0, \cdot) : [0, \infty) \to \mathbb{D}$ denote the system trajectory starting from the initial point $x_0 \in \mathbb{D}$. In other words, $x(t) = \xi(x_0, t)$ satisfies* (1) *with $x(0) = x_0$ for $t \geq 0$. Let $\xi^{-1}(x_1, \cdot) : [0, \infty) \to \mathbb{D}$ denote the backward time system trajectory starting from $x_1 \in \mathbb{D}$, so that $x(t) = \xi^{-1}(x_1, -t)$ is a solution to* (1) *with $x(0) = x_1$ for $t \leq 0$.*

By the uniqueness of solution to (1), given $x_0, x_1 \in \mathbb{D}$ and $t > 0$ such that $\xi(x_0, t) = x_1$, we have the inverse relation $\xi^{-1}(x_1, t) = x_0$. We now adopt the definitions of sensitivity and inverse sensitivity from [29] as shown in Figure 2.

Definition 2. *Given an initial state x_0, vector v, and time t, the sensitivity $\Phi(x_0, v, t)$ for the system is defined as*

$$\Phi(x_0, v, t) = \xi(x_0 + v, t) - \xi(x_0, t). \tag{2}$$

We extend the definition of sensitivity to backward time trajectories, denoted by inverse sensitivity, as

$$\Phi^{-1}(x_t, v, t) = \xi^{-1}(x_t + v, t) - \xi^{-1}(x_t, t). \tag{3}$$

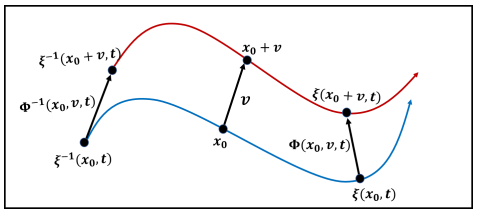


Figure 2. Visual description of the sensitivity functions Φ and Φ^{-1}. The blue and red curves, respectively, denote the unique trajectories that pass through the state of interest x_0 and its displaced state x_0+v. The sensitivity $\Phi(x_0, v, t)$ is the displacement between the respective states that the system reaches at time $t > 0$, when starting from states $x_0 + v$ and x_0 at time $t = 0$. The inverse-sensitivity $\Phi^{-1}(x_0, v, t)$ is the displacement between the states at $t = 0$ that will reach $x_0 + v$ and x_0, respectively, at time $t > 0$.

Informally, sensitivity is the vector difference between states starting from x_0 and $x_0 + v$ after time t; whereas, inverse sensitivity is the perturbation of the initial state required to displace the state at time t by v. In this work, we will primarily focus on using the inverse sensitivity function Φ^{-1} for performing systematic state space exploration, but an analogous analysis can also be conducted with the sensitivity function Φ.

For a smooth inverse-sensitivity function $\Phi^{-1}(x_0, v, t)$, let $\nabla_v \Phi^{-1}$ denote its Jacobian matrix when considered a function only of its second argument v. Then under smoothness assumption, we have the Taylor expansion

$$\Phi^{-1}(x_0, v, t) = \nabla_v \Phi^{-1}(x_0, 0, t)v + o(\|v\|) \tag{4}$$

since $\Phi^{-1}(x_0, 0, t) = 0$. Therefore learning the inverse-sensitivity function for very small v is akin to learning its directional derivative in the direction v.

A. Learning the inverse sensitivity function using observed trajectories

For testing the system operation on the domain $\mathbb{D}$, one may wish to generate a finite set of trajectories. Often, these trajectories are generated using numerical ODE solvers which return system simulations sampled at a regular time step. The step size, time bound, and the number of trajectories are specified by the user. Given a sampling of a trajectory with step size h, i.e., $\xi(x_0, 0)$, $\xi(x_0, h)$, $\xi(x_0, 2h)$, $\ldots$, $\xi(x_0, kh)$, we make a few observations. First, any prefix of this sequence is also a trajectory of a shorter duration. Hence, from a

given set of trajectories, one can truncate them to generate more trajectories having shorter duration. Second, given two trajectories starting from states x_0 and x_0', ($x_0 \neq x_0'$), we can compute the following values for the sensitivity functions:

$$\Phi(x_0, x_0' - x_0, t) = \xi(x_0', t) - \xi(x_0, t) = x_t' - x_t \quad (5)$$

$$\Phi^{-1}(x_t, x_t' - x_t, t) = x_0' - x_0 \quad (6)$$

Note that we can estimate values of Φ^{-1} based only on samples from a forward simulator ξ.

Let us explain how we generate values of the function $\Phi^{-1}(x_t, v, t)$ ($\Phi(x_0, v, t)$) for small values of v in order to learn an approximator $N_{\Phi^{-1}}(x_t, v, t)$ (or $N_{\Phi}(x_0, v, t)$). First, we generate a set of reference trajectories from initial states sampled uniformly at random. Then a fixed number of additional trajectories within a small neighborhood (with radius $\|v\| \ll 1$) of each initial state are generated. Now, we compute prefixes of the reference and its neighboring trajectories and use Equations 5 and 6 for generating tuples $\langle x_0, v, t, v_+ \rangle$ and $\langle x_t, v, t, v_- \rangle$ such that $v_+ = \Phi(x_0, v, t)$ and $v_- = \Phi^{-1}(x_t, v, t)$. We use these tuples to train either a forward sensitivity approximator denoted as N_Φ or an inverse sensitivity approximator $N_{\Phi^{-1}}$. Further details on the training procedure for learning the neural networks used in this work are mentioned in Section V-A. The training performance for various benchmark systems and neural network architectures is detailed in Section VI-A.

IV. State Space Exploration using Local Inverse Sensitivity Approximators

In this section, we show how to use an inverse sensitivity approximator $N_{\Phi^{-1}}(x_t, v, t)$ for small values of $\|v\|$ in order to perform systematic state space exploration. State space exploration is typically aimed at finding trajectories that may satisfy or violate a given specification. We primarily concern ourselves with safety specifications where the unsafe set is specified as a convex polytopes. In this setup, we would like to find trajectories that reach the set of unsafe states at a specified time, or within a certain time interval. We begin with a sub-routine for state space exploration approach to reach a given destination. We extend this method to a set of states in a subsequent section.

A. Reaching a destination at specified time

In the course of state space exploration, the designer might want to explore the system behavior that reaches a given destination or approaches the boundary condition for safe operation. Given a domain of operation, and a sample trajectory ξ, the control system designer desires to generate a trajectory that reaches a destination state z (with an error threshold of δ) at time t. Using previous notation, our goal is to find a state x such that the state $\xi(x, t)$ lies in the δ-neighborhood of z.

A toy illustration of our the state space exploration technique with oracle access to the exact inverse sensitivity function is shown in Figure 3. Given an initial point x_0, a destination z, and time t, we successively move the initial point in small steps in the direction specified by Φ^{-1}, so that the trajectory starting from the new initial point at time t moves closer to that target z with each step. In practice, since the exact inverse sensitivity function is unknown, we use a neural-network based approximation instead.

Formally, given an *anchor* trajectory starting from initial state $x_0 \in \theta$ (typically chosen at random), we first compute the vector $w^0 \doteq z - x_t^0$ where $x_t^0 \doteq \xi(x_0, t)$. Next, we estimate the inverse sensitivity $\hat{v}_-^0 \doteq N_{\Phi^{-1}}(x_t^0, sw^0, t)$ required at x_0 to move x_t towards z, and then move x_0 by $\hat{v}_-^0$. Here, the input $s \in (0, 1)$, called as the *scaling factor*, controls the magnitude of movement at each step. This process is again repeated: move the new initial state $x_0^1 \doteq x_0 + \hat{v}_-^0$ by the vector $\hat{v}_-^1 = N_{\Phi^{-1}}(x_t^1, sw^1, t)$, where $w^1 \doteq z - x_t^1$ and $x_t^1 \doteq \xi(x_0^1, t)$ is the point reached at time t by a new simulated trajectory for the system starting from initial state x_0^1. This process is repeated until x_t^k reaches a pre-specified neighborhood of z.

Since $N_{\Phi^{-1}}$ is only an approximation of Φ^{-1}, the repeated application of the former will typically compound the approximation error. Hence periodically simulating system trajectories starting from intermediate initial states – a step that we term *course correction* – is important to keep the exploration on track. Course correction steps not only confirm that the estimates at time t of the trajectory are indeed close to the z, but they also allow our procedure to make suitable adjustments if that is not indeed the case.

Since system simulation is expensive, our framework allows for course correction to be performed as frequently as desired. The parameter p is designated as *correction period* because the new anchor trajectory attempts to correct the course once for every p invocations of $N_{\Phi^{-1}}(\cdot)$. Figure 4 shows the effect of performing course corrections after every 4 steps which reduces the number of course corrections to 7 from 23 if we corrected the course at every step. Algorithm 1, which we call Reach Destination (abbreviated as $\mathcal{RD}$), provides further details of the our procedure. After termination, algorithm

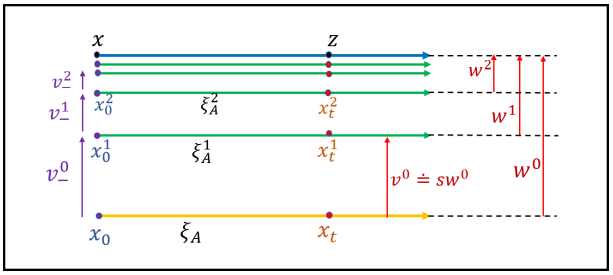


Figure 3. Toy execution of Algorithm 1 when our system consists of a constant horizontal vector field in $\mathbb{R}^2$. Suppose we are given a reference point $x_0 \in \mathbb{R}^2$, the forward simulator ξ, the target point $z \in \mathbb{R}^2$ and a time instance $t > 0$. The objective is to find the point $x \in \mathbb{R}^2$, starting from which the system reaches the point z at time t (i.e. $x \doteq \xi^{-1}(z, t)$). Starting from the initial point $x_0^0 \doteq x_0$, Algorithm 1 successively (for $i = 0, 1, 2 \ldots$) increments x_0^i by $\nu_-^i \doteq \Phi^{-1}(x_t^i, s(z - x_t^i), t)$ for a fixed $s \in (0, 1)$, where $x_t^i \doteq \xi(x_0^i, t)$ is obtained by simulating a new anchor trajectory (denoted by ξ_A^i) starting at x_0^i. In this toy example, Φ^{-1} can be calculated exactly and we have used $p = 1$ and $s = 0.5$ in Algorithm 1, but more generally, a local approximator $N_{\Phi^{-1}}$ can be used instead of Φ^{-1}, and the anchor trajectories only need to be calculated once for every $p \geq 1$ steps. The geometric nature of convergence is still preserved under these conditions (Section V).

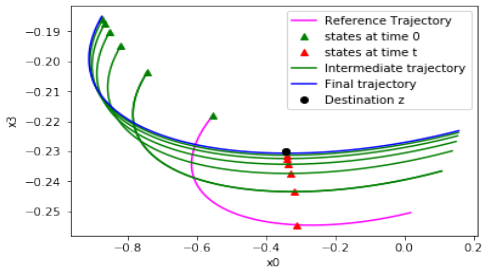


Figure 4. Correcting the course of exploration at specified period p (i.e., simulating a new trajectory after every p steps to aid the search).

input : simulator: ξ, time instance: $t \leq T$, reference trajectory: ξ_A, destination: $z \in \mathbb{D}$, course corrections bound: $\mathcal{B}$, function $N_{\Phi^{-1}}$ that approximates Φ^{-1}, initial set: θ, correction period: p, scaling factor: s, and threshold: δ.
output: course corrections: k, final trace: $\xi(x_0^k, \cdot)$, final distance: d_a^k, final relative distance: d_r

$x_0^0, x_t^0 \leftarrow \xi_A(0), \xi_A(t)$; // states at time 0 and t
$w^0 \leftarrow z - x_t^0$; // initial vector difference with z
$d_{init} \leftarrow d_a^0 \leftarrow \|w^0\|$; // initial distance
$k \leftarrow 0$;
while $(d_a^k > \delta)$ & $(k < \mathcal{B})$ **do**
  $v^k \leftarrow s \times w^k$;
  for $1 \leq j \leq p$ **do**
    $\hat{v}_-^k \leftarrow N_{\Phi^{-1}}(x_t^k, v^k, t)$; // predict v_-^k
    $x_0^k \leftarrow \hat{x}_0^{k,\theta} \leftarrow proj_\theta(x_0^k + \hat{v}_-^k)$; // perturb x_0^k
    $x_t^k \leftarrow x_t^k + v^k$; // progress x_t^k
  end
  $x_0^{k+1} \leftarrow x_0^k$;
  $\xi_A^{k+1} \leftarrow \xi(x_0^{k+1}, \cdot)$; // new anchor
  $x_t^{k+1} \leftarrow \xi_A^{k+1}(t)$; // course correction
  $w^{k+1} \leftarrow z - x_t^{k+1}$; // new vector difference
  $d_a^{k+1} \leftarrow \|w^{k+1}\|$; // update distance to z
  $k \leftarrow k + 1$; // increment corrections by 1
end
$d_r \leftarrow d_a^k / d_{init}$; // update relative distance
return (k, ξ_A^k, d_a^k, d_r);

Algorithm 1: $\mathcal{RD}$ algorithm aims at finding a trajectory that reaches δ-neighborhood of z at time t. It estimates the inverse sensitivity at each step to perturb the initial state, generates a new simulation from perturbed state after every p steps, and treats this simulation as the new anchor. k is the number of simulations generated.

$\mathcal{RD}$ returns a 4-tuple consisting of: the number of course corrections k, the trace ξ_A^k of the last anchor trajectory, the absolute distance d_a^k between the target z and ξ_A^k(t), and the relative distance d_r.

Notice that the number of course corrections is same as the number of trajectories (simulations) generated. If we were to consider physically simulating the plant (which can be expensive) as a part of the operational cost, it would make sense to minimize the number of trajectories we simulate. Limiting the number of simulated trajectories also makes the exploration algorithm more user-friendly by saving time. Thus, we choose the number of course corrections as the primary metric for performance evaluation.

V. Theoretical analysis of the convergence of ReachDestination

We now discuss the convergence of Algorithm 1. As seen in Figure 3, the distance between x_t^i and the target z contracts by a factor of $0 < 1 - sp < 1$ in each iteration if the exact inverse sensitivity is used. That is,

$$\|x_t^k - z\| \leq (1 - sp)^k \|x_t^0 - z\| \tag{7}$$

Hence, the generated trajectory will reach the desired destination within an error of δ after k^* iterations where,

$$k^* = \left\lceil \frac{\log(\|x_t^0 - z\|/\delta)}{-\log(1 - sp)} \right\rceil. \tag{8}$$

However, in the $\mathcal{RD}$ algorithm, instead of the exact inverse sensitivity function, we only use its approximation. In this section, we show that it is possible to achieve a similar geometric rate of convergence even with an approximation. Note however that the convergence of $\mathcal{RD}$ can fail badly in cases when the system is chaotic or the approximation error is large. To this end we now make assumptions on the regularity of the system and the magnitude of the approximation error that will allow for performance guarantees for $\mathcal{RD}$.

Assumption 1. *Suppose there are functions $\eta_1, \eta_2 : [0, T] \to [0, \infty)$ so that*

$$\eta_1(t)\|x - x'\| \leq \|\xi(x, t) - \xi(x', t)\| \leq \eta_2(t)\|x - x'\| \tag{9}$$

for each $x, x' \in \mathbb{D}$ and $t \in [0, T]$.

The functions η_1 and η_2, sometimes called as witnesses to the discrepancy function [16], provide worst-case bounds on how much the distance between trajectories expand or contract starting from different initial states. These functions (and their ratios) can be considered as a measure of the regularity of the system. Although in practice it may be hard to obtain the values η_1 and η_2 for the system at hand, exponential lower bound for η_1 and a similar upper bound for η_2 can be obtained using Grönwall's inequality under a Lipschitz continuity assumption on the vector field. As shown in the following lemma, Assumption 1 also ensures that $\Phi^{-1}(x, v, t)$ is a Lipschitz function of its inputs x and v. This is important as such functions can be approximated by Neural networks of bounded depth (see e.g. [30, Theorem 4.5]).

Lemma 1. *If Assumption 1 is satisfied then for any $t \in [0, T]$*

$$\|\Phi^{-1}(z', v', t) - \Phi^{-1}(z, v, t)\| \leq (2\|z - z'\| + \|v - v'\|)/\eta_1(t)$$

Proof. By taking $(x, x') = (\xi^{-1}(y, t), \xi^{-1}(y', t))$ in Assumption 1, note that $\|\xi^{-1}(y, t) - \xi^{-1}(y', t)\| \leq \|y - y'\|/\eta_1(t)$ for any $y, y' \in \mathbb{D}$. The Lemma now follows by suitably applying triangle inequality and the definiton of Φ^{-1}. □

In general, we will use the following model to measure the approximation error of $N_{\Phi^{-1}}$. The separate roles played by

the relative error ε_{rel} and the absolute error ε_{abs} will become more clear in the context of Theorem 1.

Definition 3. *$N_{\Phi^{-1}}$ is called an $(\varepsilon_{rel}, \varepsilon_{abs})$-approximator of Φ^{-1} upto radius r and time T if*

$$\|N_{\Phi^{-1}}(x_t, v, t) - \Phi^{-1}(x_t, v, t)\| \leq \varepsilon_{rel}\|\Phi^{-1}(x_t, v, t)\| + \varepsilon_{abs}$$

for any $x_t \in \mathbb{D}$, $t \in [0, T]$ and $v \in \mathbb{R}^n$, with $\|v\| \leq r$.

We are now ready state Theorem 1 which bounds the distance between z and iterate x_t^k in the kth iterations of the outer loop in $\mathcal{RD}$ when the systejm satisfies Assumption 1 and $N_{\Phi^{-1}}$ satisfies Definition 3 with sufficiently small error terms $(\varepsilon_{\text{rel}}, \varepsilon_{\text{abs}})$. To further interpret Theorem 1, note that:

1) When the additive error $\varepsilon_{\text{abs}} \approx 0$ is negligible and the relative error satisfies $\varepsilon_{\text{rel}} \in [0, \eta_1(t)\eta_2(t)^{-1}]$, Equation 10 holds for any $k \in \mathbb{N}$ with $r_{\boldsymbol{\varepsilon}}(t)/s \approx 0$. Hence, in this case, a geometric convergence similar to that described for the toy example from above continues to hold with a slightly slower convergence rate (i.e. $-\log(1 - sp\gamma_{\boldsymbol{\varepsilon}}(t))$ instead of $-\log(1 - sp)$).
2) On the other hand, when ε_{abs} is non-negligible (but sufficiently small so that $r_{\boldsymbol{\varepsilon}}(t) \leq r$), the last term in Equation 10 cannot be ignored. In this case, if rest of the assumptions of Theorem 1 are also satisfied, one obtains the guarantee that $\lim_{k\to\infty} d_a(k) \leq r_{\boldsymbol{\varepsilon}}(t)/s$. Hence if $r_{\boldsymbol{\varepsilon}}(t)/s < \delta$, the termination condition $x_t^k \in B_\delta(z)$ will eventually be satisfied whenever $k \geq k^* \doteq \lceil \log(\frac{\delta - r_{\boldsymbol{\varepsilon}}(t)/s}{d_{init}}) / \log(1 - sp\gamma_{\boldsymbol{\varepsilon}}(t)) \rceil$.

Theorem 1 (Convergence of $\mathcal{RD}$)**.** *Fix the domain $\mathbb{D} = \mathbb{R}^d$ and a time $T > 0$. Suppose*

1) *The system ξ satisfies Assumption 1,*
2) *$N_{\Phi^{-1}}$ is an $(\varepsilon_{rel}, \varepsilon_{abs})$-appoximation of Φ^{-1} for radius r and time T, and*
3) *$\varepsilon_{rel}, \varepsilon_{abs} \geq 0$ values small enough so that for each $t \in [0, T]$, $\gamma_{\boldsymbol{\varepsilon}}(t) \doteq 1 - \varepsilon_{rel}\eta_2(t)\eta_1(t)^{-1} > 0$ and $r_{\boldsymbol{\varepsilon}}(t) \doteq \varepsilon_{abs}\eta_2(t)/\gamma_{\boldsymbol{\varepsilon}}(t) \leq r$.*

Suppose the inputs $\theta = \mathbb{D}$, $t \in [0, T]$, the correction period p and the destination $z \in \mathbb{D}$ to Algorithm 1 are given and the scaling factor satisfies $s \in [r_{\boldsymbol{\varepsilon}}(t)/d_{init}, \min(r/d_{init}, 1/p)]$, where $d_{init} \doteq \|x_t^0 - z\|$ is the distance between the point $x_t^0 \doteq \xi(x_0^0, t)$ and the destination z. Then the distance after k iterations of the outer loop in Algorithm 1 satisfies the following bound

$$\|x_t^k - z\| \leq (1 - sp\gamma_{\boldsymbol{\varepsilon}}(t))^k d_{init} + \frac{r_{\boldsymbol{\varepsilon}}(t)}{s} \tag{10}$$

for any $k \in \mathbb{N}$.

The proof of Theorem 1 can be seen to be a suitable contraction argument. Formal details are given below.

Proof of Theorem 1. In this proof, for mathematical clarity, we slightly change the notation for the variables used in Algorithm 1. For each $k \geq 0$, let $x_0(k), x_t(k), w(k)$ and $d_a(k)$ denote the values of the variables x_0^k, x_t^k, w^k and d_a^k after k executions of the outer loop in Algorithm 1. Hence the equalities $w(k) = z - x_t(k)$, $d(k) = \|w(k)\|$, and $x_t(k) = \xi(x_0(k), t)$ are satisfied for any $k \geq 0$.

Since $\theta = \mathbb{D}$, unwinding the inner loop in Algroithm 1, note

$$x_0(k+1) = x_0(k) + \sum_{l=1}^{p} N_{\Phi^{-1}}(x_t(k) + (l-1)sw(k), sw(k), t). \tag{11}$$

Let $\tilde{y}(k) \doteq x_t(k+1) - x_t(k)$ denote the increment between the trajectory end points between the k and $(k+1)$th iteration. Using the fact that $x_t(k) = \xi(x_0(k), t)$, note

$$\tilde{y}(k) = \xi\left(x_0(k+1), t\right) - \xi(x_0(k), t). \tag{12}$$

The quantity $\tilde{y}(k)$ approximates the true target increment given by

$$y(k) \doteq \xi\left(x_0(k) + \Phi^{-1}(x_t(k), spw(k), t), t\right) - \xi(x_0(k), t), \tag{13}$$

which, using Definition (3) of Φ^{-1}, satisfies

$$\begin{aligned} y(k) &= \xi\left(x_0(k) + \xi^{-1}(x_t(k) + spw(k), t) - \xi^{-1}(x_t(k), t), t\right) - \xi\left(x_0(k), t\right) \\ &= \xi\left(x_0(k) + \xi^{-1}(x_t(k) + spw(k), t) - x_0(k), t\right) - x_t(k) \\ &= x_t(k) + spw(k) - x_t(k) = spw(k). \end{aligned} \tag{14}$$

Subtracting (13) from (12), and using the upper bound from (9)

$$\begin{aligned} &\|\tilde{y}(k) - y(k)\| = \left\|\xi\left(x_0(k+1), t\right) - \xi\left(x_0(k) + \Phi^{-1}(x_t(k), spw(k), t), t\right)\right\| \\ &\leq \eta_2(t)\|x_0(k+1) - x_0(k) - \Phi^{-1}(x_t(k), spw(k), t)\| \\ &= \eta_2(t)\left\|\sum_{l=1}^{p} N_{\Phi^{-1}}(x_t(k) + (l-1)sw(k), sw(k), t)\right. \\ &\qquad \left. - \sum_{l=1}^{p} \Phi^{-1}(x_t(k) + (l-1)sw(k), sw(k), t)\right\| \\ &\leq p\eta_2(t) \max_{l=1,\ldots,p} \|N_{\Phi^{-1}}(x_t(k) + (l-1)sw(k), sw(k), t) \\ &\qquad - \Phi^{-1}(x_t(k) + (l-1)sw(k), sw(k), t)\| \end{aligned} \tag{15}$$

where the equality is the third line is obtained by using (11) and rewriting $\Phi^{-1}(x_t(k), spw(k), t)$ as a telescoping sum. To bound the terms under the maximum in (15), we now use that $N_{\Phi^{-1}}$ is an $(\varepsilon_{\text{rel}}, \varepsilon_{\text{abs}})$-approximator of Φ^{-1}.

By an application of Definition 3 followed by the lower bound in (9), we obtain

$$\begin{aligned} \|N_{\Phi^{-1}}(x, v, t) - \Phi^{-1}(x, v, t)\| &\leq \varepsilon_{\text{rel}}\|\Phi^{-1}(x, v, t)\| + \varepsilon_{\text{abs}} \\ &\leq \varepsilon_{\text{rel}}\eta_1(t)^{-1}\|v\| + \varepsilon_{\text{abs}} \end{aligned} \tag{16}$$

as long as $x \in \mathbb{D}$, $\|v\| \leq r$ and $t \in [0, T]$.

Our assumptions imply that $sd_{init} \leq r$. Hence, whenever $\|w(k)\| \leq d_{init}$, we have

$$\|\tilde{y}(k) - y(k)\| \leq sp\|w(k)\|\varepsilon_{\text{rel}}\eta_2(t)\eta_1(t)^{-1} + p\eta_2(t)\varepsilon_{\text{abs}} \tag{17}$$

We will now use the above estimate to obtain a contraction-like argument. From Equation 14 we have

$$\begin{aligned} w(k+1) - w(k) &= -x_t(k+1) + x_t(k) \doteq -\tilde{y}(k) = -y(k) + y(k) - \tilde{y}(k) \\ &= -spw(k) + y(k) - \tilde{y}(k). \end{aligned}$$

Note $\|w(k)\| = \|z - x_t(k)\| = d_a(k)$. Combining the above display with Equation 17 establishes the following recursive inequality for $d_a(k)$ whenever $d_a(k) \leq d_{init}$:

$$\begin{aligned} d_a(k+1) &= \|w(k+1)\| = \|(1 - sp)w(k) + y(k) - \tilde{y}(k)\| \\ &\leq (1 - sp)\|w(k)\| + \|\tilde{y}(k) - y(k)\| \\ &\leq (1 - sp\{1 - \varepsilon_{\text{rel}}\eta_2(t)\eta_1(t)^{-1}\})d_a(k) + p\eta_2(t)\varepsilon_{\text{abs}} \\ &= (1 - sp\gamma_{\boldsymbol{\varepsilon}}(t))d_a(k) + p\eta_2(t)\varepsilon_{\text{abs}} \end{aligned}$$

where we have used the assumption $sp \leq 1$ in second line, (17) in the third line, and $\gamma_{\boldsymbol{\varepsilon}}(t) = 1 - \varepsilon_{\text{rel}}\eta_2(t)\eta_1(t)^{-1}$ in the fourth line. From the assumed lower bound on s, we have $p\eta_2(t)\varepsilon_{\text{abs}} \leq sp\gamma_{\boldsymbol{\varepsilon}}(t)d_{init}$, and the hence the condition

$d_a(k) \leq d_{init}$ continues to holds for each $k \in \mathbb{N}$ by induction. By repeatedly applying the above inequality, we obtain

$$\begin{aligned} d_a(k) &\leq (1 - sp\gamma_{\boldsymbol{\epsilon}}(t))^k d_a(0) + p\eta_2(t)\varepsilon_{\text{abs}} \sum_{i=0}^{k-1} (1 - sp\gamma_{\boldsymbol{\epsilon}}(t))^{k-1} \\ &\leq (1 - sp\gamma_{\boldsymbol{\epsilon}}(t))^k d_{init} + \frac{r_{\boldsymbol{\epsilon}}(t)}{s}. \end{aligned} \tag{18}$$

Since $sp\gamma_{\boldsymbol{\epsilon}}(t) \in [0, 1)$, we used the formula for the infinite geometric sum to obtain the last inequality. □

A. Guidance on designing better approximators

Theorem 1 also provides guidance on how to design good approximators to use with $\mathcal{RD}$. For various approximators which one may consider that satisfy Definition 3, ε_{abs} will typically be non-zero. Therefore, Theorem 1 suggests that approximators with small additive error ε_{abs} will have better reachability guarantees when used within $\mathcal{RD}$. This naturally raises the question of how to design approximators with a small additive error ε_{abs}. One important aspect of this is the evaluation radius $r > 0$. For any given approximator $N_{\Phi^{-1}}$, the additive error $\varepsilon_{\text{abs}} = \varepsilon_{\text{abs}}(r)$ in Definition 3 can be considered as an increasing function of the evaluation radius r. Therefore, one may hope to obtain estimators with better values of $\varepsilon_{\text{abs}}(r)$ by evaluating for small values of the radius r.

In Figure 5, we used test trajectories (i.e. system trajectories generated independently of the training data) to empirically estimate the absolute error $\varepsilon_{\text{abs}}(r)$ for the approximator used in NeuralExplorer evaluated for various values of r on four systems. Even as $r \to 0$, the $\varepsilon_{\text{abs}}(r)$ values of NeuralExplorer seem to approach a non-zero value $\delta_0 = \lim_{r\to 0} \varepsilon_{\text{abs}}(r) > 0$. In the light of Theorem 1, this might explain the lack of convergence of NeuralExplorer that we have observed in certain empirical examples. Indeed, as the iterates x_t^k in the NeuralExplorer approach the target z, the error in the approximation of $N_{\Phi^{-1}}$ might possibly be dominating the increment $\Phi^{-1}(x_t, s(z - x_t), t)$ needed to proceed towards the target.

Motivated by this discussion, in this work, we introduce approximators based on neural networks $\tilde{N}_{\Phi^{-1}}(x_t, v/\|v\|, t)$ that learn only the direction (and not the magnitude) of the vector $\Phi^{-1}(x_t, v, t) \approx \nabla_v \Phi^{-1}(x_t, 0, t)v$, for small values of $\|v\|$. By focusing only on learning the direction, we avoid numerical issues involved in learning small vectors. Intuitively, if the value $\|\Phi^{-1}(x_t, v, t)\|$ was then known, we could use the oracle-estimator

$$N_{\Phi^{-1}}(x_t, v, t) = \tilde{N}_{\Phi^{-1}}(x_t, \frac{v}{\|v\|}, t)\|\Phi^{-1}(x_t, v, t)\| \tag{19}$$

to approximate $\Phi^{-1}(x_t, v, t)$ for small values of $\|v\|$. Figure 5 shows the estimate of the $\varepsilon_{\text{abs}}(r)$ versus r plot for the oracle estimator in (19). However, note that outside a testing scenario like that in Figure 5, $N_{\Phi^{-1}}(x_t, v, t)$ cannot be evaluated for the directional approximators. Instead, we directly use $\tilde{N}_{\Phi^{-1}}(x_t, \frac{v}{\|v\|}, t)$ in $\mathcal{RD}$ algorithm by the modifications mentioned in Remark 1.

Remark 1. *As mentioned above, it may be helpful to work with directional-approximators $\tilde{N}(x_t, v/\|v\|, t)$ for $\Phi^{-1}(x_t, v, t)$, which learn only the direction (and not the magnitude) of the vector $\Phi^{-1}(x_t, v, t) \approx \nabla_v \Phi^{-1}(x_t, v, t)v$ for small values of $\|v\|$. One may then work directly with the the directional approximator $\tilde{N}(x_t, v/\|v\|, t)$ in $\mathcal{RD}$ by simply replacing line 8 with the two statements*

$(8a)$ $\hat{v}_-^k \leftarrow \tilde{N}_{\Phi^{-1}}(x_t^k, \frac{v^k}{\|v^k\|}, t)$

$(8b)$ $\hat{v}_-^k \leftarrow (s \times \|v^k\|) \cdot \hat{v}_-^k$,

while keeping the rest of the algorithm unchanged.

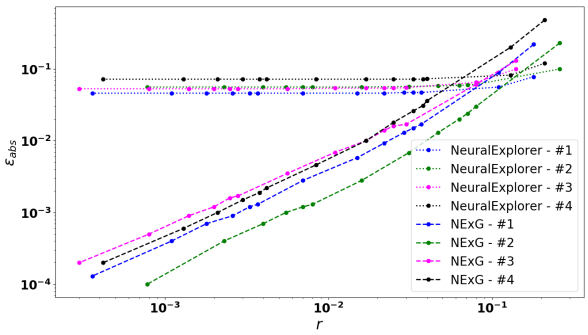


Figure 5. Empirical values of the additive error ε_{abs} in Definition 3 (assuming $\varepsilon_{\text{rel}} \approx 0$) for the approximators $N_{\Phi^{-1}}$ learned for NeuralExplorer and NExG as a function of the evaluation radius r, for four benchmark systems. Note that we have the used the oracle-estimator $N_{\Phi^{-1}}$ given by (19) to estimate the additive error of NExG, since NExG only learns a directional approximator $\tilde{N}_{\Phi^{-1}}$.

The NExG algorithms used in the subsequent sections are based on directional-approximators $\tilde{N}_{\Phi^{-1}}$ and the corresponding modifications to $\mathcal{RD}$ as mentioned in Remark 1.

VI. Evaluation

We choose a standard benchmark suite of control systems with neural feedback functions [19], [34], [35], [41] for evaluation. To be more specific, systems #10-#12 are adopted from [34], system #13 is adopted from [19] and the rest of the benchmarks are adopted from the ARCH suite [35], [41]. Considered benchmarks span 6 dimensional systems, controllers with 6-10 hidden layers and 100-300 neurons per layer (c.f. Table I). The tool along with the learned models and presented artifacts is available at https://github.com/manishgcs/NExG.

A. Network architecture and Training

For each benchmark, we sample a fixed number (40) of initial states chosen uniformly at random and use a given ODE solver to generate *anchor* trajectories from these initial states. We further generate ten additional trajectories from the states randomly sampled in the small neighborhood ($\|v\| = 0.01$) of each initial state. We choose a step size ($h = 0.01$). Max step (T) for each system is shown in Table I, however, a user can pick any suitable values for these parameters including the number of anchor trajectories. Our preliminary analysis shows that increasing the number of *anchor* trajectories from 40 to 50 slightly improves the MRE, however, this improvement comes at the expense of higher training time. These trade offs between the amount of data required, training time,

distance between neighboring points, and accuracy of the approximation are subjected to future research. Nonetheless, the evaluations presented in subsequent sections underscore the promise of our approach even when the resources are constrained. The data used for training the neural network is collected as previously described. We use 90% of the data for training and 10% for testing.

TABLE I

TRAINING $N_{\Phi^{-1}}$. EACH NEURAL NETWORK FEEDBACK CONTROLLER CONFIGURATION IS GIVEN AS THE *number of hidden layers* AND THE MAXIMUM OF *neurons per layer*. *Dims* IS THE NUMBER OF SYSTEM VARIABLES AND T IS SIMULATION TIME BOUND. THE TRAINING PERFORMANCE IS MEASURED IN *mean squared error* (MSE) AND *mean relative error* (MRE).

System				$N_{\Phi^{-1}}$ Training			
No.	Name	Dims	NN controller config	Max steps (T)	Training Time (min)	MSE	MRE %
#1	ARCH-1	2	6/50	200	8.0	0.018	16.0
#2	ARCH-2	2	7/100	300	11.0	0.038	15.0
#3	ARCH-3	2	5/50	350	14.0	0.014	10.2
#4	S. Pend.	2	2/25	250	9.0	0.021	15.0
#5	ARCH-4	3	7/100	250	10.0	0.007	6.0
#6	ARCH-5	3	7/100	250	10.0	0.009	13.0
#7	ARCH-6	3	6/100	250	10.0	0.007	5.6
#8	ARCH-7	3	2/300	250	11.0	0.01	9.5
#9	ARCH-8	4	5/100	250	12.0	0.005	8.0
#10	ARCH-9-I	4	3/100	250	12.0	0.005	4.2
#11	ARCH-9-II	4	3/20	250	11.0	0.005	7.3
#12	ARCH-9-III	4	3/20	250	11.0	0.005	4.5
#13	Unicycle	4	1/500	250	11.0	0.005	5.3
#14	D. Pend.-I	4	2/25	250	11.0	0.006	8.1
#15	D. Pend.-II	4	2/25	200	10.0	0.02	17.0
#16	I. Pend.	4	1/10	200	9.0	0.007	9.0
#17	ACC-3L	6	3/20	200	11.0	0.004	8.3
#18	ACC-5L	6	5/20	200	11.0	0.004	6.7
#19	ACC-7L	6	7/20	200	12.0	0.004	6.7
#20	ACC-10L	6	10/20	200	12.0	0.005	12.0

We use Python Multilayer Perceptron implemented in `Keras 2.3` [9] library with Tensorflow as the backend. Every network has 3 layers with 512 neurons each and an output layer. The input layer's activation function is *Radial Basis Function* (RBF) with *Gaussian basis* [55]. The other two layers have `ReLU` activation (except System #11 which has its feedback controller trained with `Sigmoid` activation) and the output layer has *linear* activation function. The optimizer used is stochastic gradient descent. The network is trained using Levenberg-Marquardt backpropagation algorithm optimizing the mean absolute error loss function and the Nguyen-Widrow initialization. The training and evaluation are performed on a system running Ubuntu 18.04 with a 2.20GHz Intel Core i7-8750H CPU with 12 cores and 32 GB RAM. The network $N_{\Phi^{-1}}$ training time, *mean squared error* (MSE) and *mean relative error* (MRE) for learning Φ^{-1} are given in Table I.

Although empirical, our choice for network architecture and evaluation metrics are motivated by our previous works [27], [29]. But their network training time and training error are notably high, which may render them unfavorable for many practical applications. After performing experiments on multiple activation functions, we choose a non linear multivariate radial basis function (RBF) with Gaussian basis as the input layer's activation function because we observe that its evaluation performance is consistent across benchmarks.

TABLE II

PERFORMANCE EVALUATION W.R.T. NEURALEXPLORER. THE COMMON PARAMETERS VALUES ARE $\delta = 0.004$ AND $\mathcal{B} = 30$. WE FIX $s = 0.5$ AND $p = 2$ FOR NEXG. k IS THE NUMBER OF SIMULATIONS GENERATED BY RESPECTIVE ALGORITHMS AND $d_r\% = (d_a^k/d_{init}) \times 100$ WHERE d_a^k IS THE DISTANCE BETWEEN THE STATE OF FINAL (k^{th}) SIMULATION AT TIME t AND THE DESTINATION.

Sys.	NeuralExp.		NExG		Sys.	NeuralExp.		NExG	
	k	$d_r\%$	k	$d_r\%$		k	$d_r\%$	k	$d_r\%$
#1	21	14	5	1.8	#11	30	6.1	4	0.4
#2	27	9.6	6	1.3	#12	30	13.8	4	0.6
#3	10	6.4	5	2.7	#13	30	11.4	13	2.3
#4	18	5.5	5	1.4	#14	29	7.6	10	1.9
#5	30	13.3	7	2.7	#15	26	12.7	8	3.7
#6	24	11.2	5	3.9	#16	29	22.5	6	2.2
#7	28	12.5	5	2.1	#17	30	8.0	8	0.4
#8	23	5.5	7	1.7	#18	30	6.7	7	0.4
#9	30	12.3	12	3.5	#19	30	5.5	6	0.3
#10	29	4.3	10	1.0	#20	30	5.4	17	1.6

B. ReachDestination Evaluation and Features

We analyze the performance of Algorithm 1 by picking, at every invocation of the algorithm, a random reference trajectory ξ_A, a time $t \in [0, T]$, and a target state z, reachable at time t in the domain of interest. We choose them randomly to not bias the evaluation of our search procedure to a specific sub-space. The performance metrics used to evaluate various runs are *number of course corrections* (k) and/or *minimum relative distance* (d_r). The threshold δ is fixed as 0.004.

B.1 Comparison with `NeuralExplorer` [29]: The neural network architectures used in this work are the same as those used in NeuralExplorer. For a given $N \in \mathbb{Z}_+$ number of anchor trajectories, NeuralExplorer creates all possible $C(N, 2)$ pairs of these trajectories for training as it attempts to learn the inverse sensitivity function for any $v \in \mathbb{R}^n$ in the domain of interest. Whereas NExG focuses on learning only the direction of the inverse sensitivity. So we only sample a few random points (say, y) in a small neighborhood of the initial state of each anchor trajectory and generate total $y \times N$ pairs. As a consequence, we achieve up to 60% reduction in the training time. Further, the state space exploration algorithm in NeuralExplorer predicts inverse sensitivity directly for w^k and course corrects at every step; whereas, NExG predicts only the direction of the inverse sensitivity vector needed to move in the direction w^k. Hence the NExG search is guided by additional parameters like the scaling factor s and the correction period p. We report in Table II the mean values of k and d_r computed over 250 runs of each technique for each system. The evaluation shows that NExG has a relative error of 1-4% (with considerably fewer iterations) as compared to the relative error of 5-15% for NeuralExplorer.

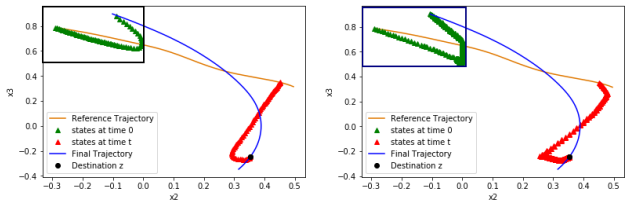
B.2 Correction period (p) and scaling factor (s): We fix the tuple (ξ_A, z, t) in each benchmark, run $\mathcal{RD}$ for $s \in \{0.01, 0.1\}$, $p \in \{1, 5, 10\}$. The evaluation results in Table III are presented to make some key observations and emphasize that the technique performs consistently across systems. For a fixed tuple (ξ_A, z, t), change in the number of trajectories (k) generated by $\mathcal{RD}$ is roughly inversely proportional to the change in the product $s \cdot p$. For example, first row (i.e., $s = 0.01$) in System #7 shows that the number of trajectories (k) reduces from ~ 400 to ~ 40 (10 fold reduction)

TABLE III
$\mathcal{RD}$ **EVALUATION**. d_{init} IS THE DISTANCE BETWEEN THE STATE OF THE INITIAL REFERENCE TRAJECTORY AT TIME t AND DESTINATION z, s IS THE SCALING FACTOR, AND p IS THE CORRECTION PERIOD.

System	d_{init}	s	Course corrections k		
			$p=1$	$p=5$	$p=10$
#7	0.39	0.01	418	83	41
		0.1	41	7	3
#17	0.85	0.01	550	109	54
		0.1	54	10	3

when course correction is performed only once for every 10 steps instead of at every steps. This trend is observed in almost all systems for appropriate $s \cdot p$ values. It can also be observed that the number of course corrections (k) remains roughly the same for different (s, p) pairs as long as the product $s \cdot p$ is same. For e.g., the value of k for pairs ($s = 0.01$, $p = 10$) and ($s = 0.1$, $p = 1$) is ~ 40 in System #1. These results are consistent with the theoretical bound (Equation 10) that decreases geometrically in k for a fixed value of $s \cdot p$.

B.3 Satisfying initial conditions: As the algorithm increments the initial state x_0^k in line 9, it may happen that next $\hat{x}_0^{k+1} \doteq (x_0^k + \hat{v}_-^k)$ is not in the initial set θ, thus violating the initial constraint. We address this problem by picking a element-wise projection of $\hat{x}_0^k$ in θ denoted as $\hat{x}_0^{k,\theta} \doteq proj_\theta(\hat{x}_0^k) \in \theta$, defined by $\hat{x}_0^{k,\theta} \doteq \arg\min_{x\in\theta}\|x - \hat{x}_0^k\|$. Consider System #10 with 2^{nd} component of its initial set hyper-rectangle, given as $\theta[2] \doteq [-0.5, 0.0]$. Both Figures 6(a) and 6(b) demonstrate how the course of exploration makes a detour around the initial set boundary in order to satisfy its constraints.



(a) Original $\mathcal{RD}$ (b) Axis-aligned $\mathcal{RD}$

Figure 6. $\mathcal{RD}$ and its customizations can provide algorithms for state space exploration with a constrained initial set. In the figure, the inner box represents the initial set. Original $\mathcal{RD}$ generates smoother trajectories because it moves in the direction of the target at each step (Figure 6(a)), however $\mathcal{RD}$ can be customized to obtain a different state space exploration method.

B.4 Customizing the state exploration algorithm: Note that the inverse sensitivity approximator $N_{\Phi^{-1}}$ is agnostic to the exact state space exploration technique. While our implementation of $\mathcal{RD}$ uses this estimator to proceed in a straight line direction towards the destination (i.e. v^k has the same direction as $z - x_t^k$), the progress direction can also be customized. This allows for designing custom state space exploration algorithm by prioritizing trajectories along different directions at different steps. For an n-dimensional system, at every step, one might be interested in picking a direction among the $2n$ unit vectors $\{\pm e_i : i = 1, 2, \ldots, n\}$ that are aligned with the orthonormal axes. For instance, one can choose the direction vector that is closest to $z - x_t$. The illustration of one such *axis-aligned* approach is given in Figure 6(b). It emphasizes that instead of $\mathcal{RD}$, we can also use some other state space exploration algorithm that requires an inverse sensitivity approximator.

B.5 Coverage analysis: Given an initial set $\theta \subseteq \mathbb{D}$, we assess the coverage among the set of reachable states at time $t \in [0, T]$ by calculating the proportion of points in the reachable set that $\mathcal{RD}$ converges to, within a neighborhood of radius δ. To obtain a convenient representation of the reachable set for an n-dimensional system, we use a polygon with faces in the $2n$ template directions $\{\pm e_i : i = 1, 2, \ldots, n\}$. While we have used orthonormal vectors as template directions, different set of template directions can yield a less conservative approximation of the reachable set. The polygon in our experiment was obtained by starting from the destination state of random anchor trajectory $\xi_A(\cdot)$ at time t, and using a modification of $\mathcal{RD}$ to maximally perturbs the destination state in each of the template directions. This provides as many extremal points as the number of template directions, and can be used to construct the bounding polygon (e.g. see the black rectangle in Figure 7), denoted by $\mathcal{Z}$, as an approximation of the reachable set. Next, to assess the coverage for $\mathcal{Z}$, we sampled 200 points from $\mathcal{Z}$ uniformly at random, and examined which were the ones that $\mathcal{RD}$ could converge within a $\delta = 4 \times 10^{-3}$ neighborhood at time t starting from the initial set θ. As shown in Figure 7, 137 out of these 200 points were reached from $\mathcal{RD}$, with the color of the point (green or red) representing if $\mathcal{RD}$ was successful or not. For each of these points in $\mathcal{Z}$, we also plot the best initial point output by $\mathcal{RD}$. Most of these red initial points (that did not reach the destination) lie on the boundary of the initial set θ, suggesting that the trajectory that can possibly reach its destination might perhaps start from a state outside the given initial set.

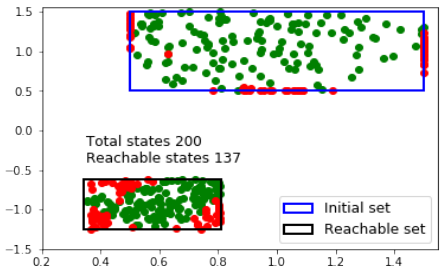


Figure 7. Measuring coverage of a reachable set in System #1. For every red colored state in the destination set, $\mathcal{RD}$ could not find a trajectory that reaches within its δ-neighborhood at time t.

VII. Falsification of a Safety Specification

Given system and a corresponding safety specification in either Signal or Metric Temporal Logic [36], [42], *falsification* is aimed at finding a system parameter or an input that violates the specification. Existing falsification schemes generate executions using some heuristics or stochastic global optimization and compute their robustness with respect to a

safety specification provided as a set of states. *Robustness* ($\rho \in \mathbb{R}$) is a measure that quantifies how deep is the execution within the set or how far away it is from the set. Informally, it determines the degree to which an execution satisfies ($\rho > 0$) or violates ($\rho < 0$) a given safety specification. Our framework can currently handle a subset of MTL formulas.

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \top \mathcal{U}_l \varphi$$

where p is an atomic proposition, l is a non-empty interval of $\mathbb{R}_+$, and φ is a well formed MTL formula. The temporal operator $\diamond$ $(eventually)$ is defined as $\diamond_l \varphi := \top \mathcal{U}_l \varphi$. The reader can refer to [46] for robust semantics of MTL formulas.

A. Our Falsification algorithm

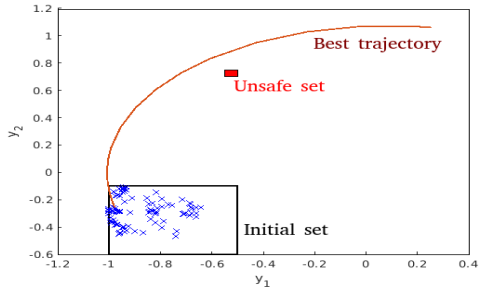
We describe a simple $\mathcal{RD}$-based algorithm to obtain a falsifying trajectory to a given safety specification $\neg\Diamond_l U$, where $U \subseteq \mathbb{R}^n$ is the unsafe set. We generate an anchor trajectory ξ_A, sample a state $z \in U$, and choose $t = \arg\min_{t' \in l} \|\xi_A(t') - z\|$. We then invoke $\mathcal{RD}$ sub-routine for generating trajectories until we obtain a counterexample ($\rho^k < 0$) to the given safety specification or bound I is exhausted, where ρ^k is the robustness of trajectory ξ_A^k. To be precise, in the falsification run of $\mathcal{RD}$- (i) distance d_a^k is replaced by robustness ρ^k, (ii) constraint $d_a^k > \delta$ is replaced by $\rho^k > 0$, and (iii) an additional constraint $x_t^k \notin U$ is added to the main **while** loop condition. While it may be the case that z is not reachable at time t, both these parameters primarily act as anchors to guide the procedure in obtaining a falsifying execution.

B. Evaluation of Falsification techniques

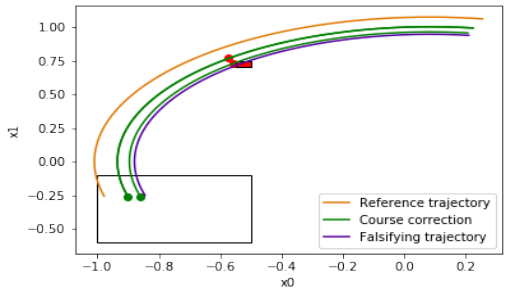
We evaluate our falsification algorithm against one of the widely used *falsification* platforms, S-TaLiRo [46]. Monte-Carlo sampling scheme in S-TaLiRo is sensitive to the "temperature" parameter β, where the adaptation of β is performed after every fixed number of iterations provided it is unable to find a counterexample by then. We keep $\beta = 50$ which is the default value, and we consider $p = 2, s = 0.5$ for our $\mathcal{RD}$-based falsification scheme. Although adaptation parameters and mechanisms in both approaches are different, an upper bound ($\mathcal{B}$) on the number of trajectories is crucial to both of them. We fix $\mathcal{B} = 100$ for systems #1-#16 and $\mathcal{B} = 150$ for systems #17-#30 in S-TaLiRo. We consider $\mathcal{B} = 50$ for NExG as we notice that, if it can, it usually finds a trajectory of interest in notably less number of iterations. The sampling time is fixed as 0.01. We exclude cases where the initial reference trajectory ξ_A is falsifying so as to minimize the bias induced by different distributions in different schemes. For a given pair of initial configuration θ and safety specification $\neg\Diamond_l U$ in each system, we report in Table IV the *mean* of total trajectories (k) generated along with *mean* robustness (ρ) computed over 250 runs of respective techniques.

The evaluations exhibit that our algorithm not only takes a very few trajectories to converge but also the counterexample obtained is relatively more robust in most cases. Unlike NExG, the performance of S-TaLiRo seems to deteriorate further with increase in the number of system dimensions and complexity. Even in scenarios where $\mathcal{RD}$ generates more than 10 trajectories, experiments indicate that it is able to reach the neighborhood around U within fewer iterations. This observation motivated us to attempt to integrate both frameworks. In the case of non-convergence in S-TaLiRo, its best execution can be used as the input *reference* trajectory for $\mathcal{RD}$. One such instance is shown in Figure 8(a) where S-TaLiRo is unable to find a falsifying trajectory within 100 iterations. We use its best sample as the reference ξ_A for $\mathcal{RD}$ and find 4^{th} trajectory to be a counterexample (Figure 8(b)). This exercise is performed for illustration purpose i.e., at present we manually port the best sample from S-TaLiRo to NExG. One of the future tasks is to automate this integration. Additionally, our approach - as a side effect - provides intuition about the course of exploration leading to the falsifying execution unlike scattered stochastically sampled states generated in S-TaLiRo.

Another important take away from this comparison is that if S-TaLiRo fails to find a counterexample for a given specification, the user is left with a sample of trajectories generated by S-TaLiRo and the execution that comes closest to falsifying the given safety specification. Instead, in our case, the user can still access the inverse sensitivity approximator and manually (or algorithmically) probe nearby trajectories and proceed to discover a falsifying trajectory. Finally, S-TaLiRo's implementation platform is `MATLAB` while our framework is implemented in `Python`. We do not report the wall-clock time taken by respective frameworks as performance differences are expected due to their different implementation platforms.



(a) from S-TaLiRo



(b) from NExG

Figure 8. Falsification demonstrations. The red-colored box is the unsafe set and the inner while-colored box is the initial set. These demonstrations depict how NExG can potentially supplement other falsification platforms if they fail to find a falsifying execution.

TABLE IV

PERFORMANCE OF FALSIFICATION TECHNIQUES. k IS THE NUMBER OF SIMULATIONS GENERATED AND ρ IS THE ROBUSTNESS. THE PARITY OF ρ DETERMINES WHETHER THE EXECUTION SATISFIES ($\rho > 0$) OR FALSIFIES ($\rho < 0$) A GIVEN SAFETY SPECIFICATION, WHEREAS ITS MAGNITUDE DETERMINES HOW ROBUST IS THE EXECUTION. NExG TAKES A VERY FEW ITERATIONS TO FIND A COUNTEREXAMPLE WITH $\rho < 0$. ✓MARKS THE SCENARIOS WITH EQUALLY (OR MORE) ROBUST FALSIFYING TRAJECTORY.

System	Initial configuration θ	Safety specification $\neg\Diamond_I U$	S-TaLiRo Mean k	S-TaLiRo Mean ρ	NExG Mean k	NExG Mean ρ
#1	[(0.5, 1.5)∧(0.5, 1.5)]	$\neg\Diamond_{[0.7,0.9]}$[(0.30, 0.35)∧(-1.1, -1.05)]	12	-0.01✓	3	-0.01✓
#2	[(0.8, 1.2)∧(0.9, 1.2)]	$\neg\Diamond_{[0.7,0.9]}$[(1.50, 1.55)∧(0.20, 0.25)]	24	-0.006✓	3	-0.005
#3	[(0.4, 1.2)∧(0.4, 1.2)]	$\neg\Diamond_{[0.8,1.0]}$[(0.3, 0.4)∧(0.0, 0.1)]	8	-0.02✓	4	-0.014
#4	[(1.5, 2.0)∧(1.0, 1.5)]	$\neg\Diamond_{[0.7,0.9]}$[(1.15, 1.2)∧(-0.95, -0.90)]	11	-0.008✓	3	-0.008✓
#5	[(0.2, 0.7)∧(0.2, 0.7)∧(0.2, 0.7)]	$\neg\Diamond_{[0.6,0.8]}$[(1.0, 1.05)∧(0.05, 0.1)∧(-1.15, -1.10)]	39	0.008	17	-0.005✓
#6	[(0.1, 0.6)∧(0.1, 0.6)∧(0.1, 0.6)]	$\neg\Diamond_{[0.7,0.9]}$[(0.10, 0.15)∧(0.0, 0.05)∧(0.10, 0.15)]	21	-0.005	3	-0.007✓
#7	[(0.2, 0.5)∧(0.2, 0.5)∧(0.2, 0.5)]	$\neg\Diamond_{[1.0,1.2]}$[(0.4, 0.45)∧(-0.3, -0.25)∧(-0.45, -0.4)]	20	-0.005	3	-0.007✓
#8	[(0.2, 0.5)∧(0.2, 0.5)∧(0.2, 0.5)]	$\neg\Diamond_{[1.0,1.2]}$[(0.05, 0.1)∧(0.25, 0.3)∧(-0.35, -0.3)]	36	0.008	9	0.001✓
#9	[(0.1, 0.4)∧(0.1, 0.4) ∧(0.1, 0.4)∧(0.1, 0.4)]	$\neg\Diamond_{[0.6,0.8]}$[(-0.15, -0.10)∧(-0.80, -0.75) ∧(0.0, 0.05)∧(-0.60, -0.55)]	76	0.005✓	22	0.005✓
#10	[(0.5, 1.0)∧(-1, -0.5) ∧(-0.5, 0)∧(0.5, 1)]	$\neg\Diamond_{[0.8,1.0]}$[(-0.2, -0.15)∧(-1.05, -1.0) ∧(0.2, 0.25)∧(0.25, 0.3)]	88	0.01	5	-0.005✓
#11	[(-1, -0.5)∧(-0.6, -0.1) ∧(0.2, 0.7)∧(-0.5, 0)]	$\neg\Diamond_{[1.0,1.2]}$[(-0.55, -0.5)∧(0.7, 0.75) ∧(0.65, 0.7)∧(0.25, 0.3)]	95	0.031	3	-0.016✓
#12	[(-1, -0.5)∧(-0.6, -0.1) ∧(0.2, 0.7)∧(-0.5, 0)]	$\neg\Diamond_{[1.0,1.2]}$[(-0.55, -0.5)∧(0.25, 0.3) ∧(-0.05, 0.0)∧(-0.3, -0.25)]	98	0.058	6	-0.009✓
#13	[(9.3, 9.7)∧(-4.7, -4.3) ∧(2, 2.4)∧(1.3, 1.7)]	$\neg\Diamond_{[0.8,1.0]}$[(8.4, 8.45)∧(-3.55, -3.5) ∧(2.5, 2.55)∧(2.2, 2.25)]	81	0.009	4	-0.01✓
#14	[(1.0, 1.5)∧(1.0, 1.5) ∧(1, 1.5)∧(1, 1.5)]	$\neg\Diamond_{[1.0,1.2]}$[(1.55, 1.6)∧(0.25, 0.3) ∧(-0.65, -0.6)∧(-1.2, -1.15)]	59	-0.002✓	6	-0.002✓
#15	[(1.0, 1.4)∧(1.0, 1.4) ∧(1.0, 1.4)∧(1.0, 1.4)]	$\neg\Diamond_{[0.4,0.6]}$[(0.90, 0.95)∧(0.65, 0.7) ∧(-1.8, -1.75)∧(-1.20, -1.15)]	55	-0.002	8	-0.007✓
#16	[(0.0, 0.3)∧(0.0, 0.3) ∧(0.0, 0.3)∧(-0.3, 0.0)]	$\neg\Diamond_{[0.8,1.0]}$[(0.05, 0.1)∧(-0.05, 0.0) ∧(-0.05, 0.0)∧(0.0, 0.05)]	47	-0.002	9	-0.003✓
#17	[(90.0, 92.0)∧(32.0, 32.5)∧(0.0, 0.0) ∧(10.0, 11.0)∧(30.0, 30.5)∧(0.0, 0.0)]	$\neg\Diamond_{[0.7,0.9]}$[(113.5, 114.0)∧(31.3, 31.4)∧(-1.60, -1.55) ∧(32.0, 32.5)∧(29.5, 30.0)∧(-0.10, -0.05)]	150	0.10	15	-0.001✓
#18	[(90.0, 92.0)∧(32.0, 32.5)∧(0.0, 0.0) ∧(10.0, 11.0)∧(30.0, 30.5)∧(0.0, 0.0)]	$\neg\Diamond_{[0.7,0.9]}$[(116.5, 117.0)∧(31.6, 31.7)∧(-1.65, -1.6) ∧(34.5, 35.0)∧(29.5, 30.0)∧(-0.45, -0.35)]	150	0.81	14	-0.002✓
#19	[(90.0, 92.0)∧(32.0, 32.5)∧(0.0, 0.0) ∧(10.0, 11.0)∧(30.0, 30.5)∧(0.0, 0.0)]	$\neg\Diamond_{[0.8,1.0]}$[(120.0, 120.3)∧(31.1, 31.2)∧(-1.75, -1.7) ∧(37.0, 38.0)∧(30.5, 31.0)∧(0.1, 0.2)]	149	0.11	9	-0.007✓
#20	[(90.0, 92.0)∧(32.0, 32.5)∧(0.0, 0.0) ∧(10.0, 11.0)∧(30.0, 30.5)∧(0.0, 0.0)]	$\neg\Diamond_{[0.6,0.8]}$[(112.8, 112.9)∧(31.7, 31.8)∧(-1.55, -1.5) ∧(31.0, 31.5)∧(30.1, 30.2)∧(0.0, 0.1)]	145	0.14	34	0.015✓

VIII. DISCUSSION AND FUTURE WORK

In this work, we have proposed a new state space exploration technique NExG that is an improvement over existing state space approaches. In addition to out-performing state of the art falsification techniques, our technique enables the control designer to develop custom algorithms for state space exploration and generate trajectories that navigate the state space along with additional constraints.

We also would suggest two additional use-cases of our approach: generating near-miss trajectory and estimated trajectories. Instead of continuing to run the $\mathcal{RD}$ algorithm to search for execution that reaches the target state, a control designer can terminate early and use a custom state space exploration using inverse sensitivity to generate *near miss* safety instances where the trajectory approaches the set of unsafe states within a threshold. Secondly, if invoking the exact simulation engine becomes computationally expensive (e.g. exploring a high-dimensional space), then the control designer can generate approximate trajectories from a given initial state by using a sensitivity estimator.

Future Work: As our choice of network architecture, training data, scaling factor and other parameters is mostly empirical, we plan to investigate different architectures to accelerate network training and explore other parameters configurations to improve performance of NExG. We aim to extend this work to handle more generic systems such as feedback systems with environmental inputs. Further, we would like to enhance our falsification technique to the general class of STL specification and automate its integration with S-TaLiRo. We finally aim to explore ways to integrate our method into frameworks used for generating adversarial executions during control synthesis.

ACKNOWLEDGMENTS

The authors would like to dedicate this paper to the memory of Oded Maler, who has been a source of inspiration and support. This paper is inspired by Oded's work on Systematic Simulation using Sensitivity Analysis. This research has been partially supported by Air Force Office of Scientific Research under award number FA9550-19-1-0288, National Science Foundation (NSF) under grant numbers CNS 2038960, and Amazon Research Award — Automated Reasoning. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force, National Science Foundation, or Amazon.

REFERENCES

[1] Houssam Abbas and Georgios Fainekos. Linear hybrid system falsification through local search. In *ATVA*, pages 503–510. Springer, 2011.

[2] Matthias Althoff. An introduction to CORA. In *ARCH@CPSWeek*, EPiC Series in Computing (34), pages 120–151. EasyChair, 2015.

[3] Rajeev Alur, Thao Dang, and Franjo Ivančic. Progress on reachability analysis of hybrid systems using predicate abstraction. In Oded Maler and Amir Pnueli, editors, *HSCC*, page 4–19. Springer-Verlag, 2003.

[4] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *TACAS*, pages 254–257, Berlin, 2011. Springer.

[5] Stanley Bak and Parasara Sridhar Duggirala. Hylaa: A tool for computing simulation-equivalent reachability for linear systems. In *HSCC*, page 173–178, NY, USA, 2017. ACM.

[6] Sergiy Bogomolov, Goran Frehse, Amit Gurung, Dongxu Li, Georg Martius, and Rajarshi Ray. Falsification of hybrid systems using symbolic reachability and trajectory splicing. In *HSCC*, 2019.

[7] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations, 2018.

[8] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *CAV*, pages 258–263, 2013.

[9] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[10] Thao Dang, Alexandre Donze, Oded Maler, and Noa Shalev. Sensitive state-space exploration. In *CDC*, pages 4049–4054, 2008.

[11] Thao Dang and Oded Maler. Reachability analysis via face lifting. In *HSCC*, pages 96–109. Springer, 1998.

[12] Jyotirmoy V. Deshmukh, Georgios Fainekos, James Kapinski, Sriram Sankaranarayanan, Aditya Zutshi, and Xiaoqing Jin. Beyond single shooting: Iterative approaches to falsification. In *ACC*, page 4098, 2015.

[13] Alexandre Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *CAV*, pages 167–170. Springer, 2010.

[14] Alexandre Donzé and Oded Maler. Systematic simulation using sensitivity analysis. In *HSCC*, pages 174–189. Springer, 2007.

[15] Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In *Formal Modeling and Analysis of Timed Systems*, pages 92–106, Berlin, Heidelberg, 2010. Springer.

[16] Parasara Sridhar Duggirala, Sayan Mitra, and Mahesh Viswanathan. Verification of annotated models from executions. In *EMSOFT*, 2013.

[17] Parasara Sridhar Duggirala, Matthew Potok, Sayan Mitra, and Mahesh Viswanathan. C2e2: A tool for verifying annotated hybrid systems. In *HSCC*, page 307–308, NY, USA, 2015. ACM.

[18] Souradeep Dutta, Xin Chen, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Sherlock - a tool for verification of neural network feedback systems: Demo abstract. In *HSCC*, page 262–263. ACM, 2019.

[19] Souradeep Dutta, Xin Chen, and Sriram Sankaranarayanan. Reachability analysis for neural feedback systems using regressive polynomial rule inference. In *HSCC*, page 157–168. ACM, 2019.

[20] Georgios E. Fainekos and George J. Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42):4262–4291, 2009.

[21] Chuchu Fan and Sayan Mitra. Bounded verification with on-the-fly discrepancy computation. In *ATVA*, pages 446–463. Springer, 2015.

[22] Chuchu Fan, Bolun Qi, Sayan Mitra, and Mahesh Viswanathan. Dryvr: Data-driven verification and compositional reasoning for automotive systems. In *CAV*, pages 441–461. Springer, 2017.

[23] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In *Computer Aided Verification*, pages 379–395. Springer, 2011.

[24] Shromona Ghosh, Dorsa Sadigh, Pierluigi Nuzzo, Vasumathi Raman, Alexandre Donzé, Alberto L. Sangiovanni-Vincentelli, S. Shankar Sastry, and Sanjit A. Seshia. Diagnosis and repair for synthesis from signal temporal logic specifications. In *HSCC*, page 31–40. ACM, 2016.

[25] Manish Goyal, David Bergman, and Parasara Sridhar Duggirala. Generating longest counterexample: On the cross-roads of mixed integer linear programming and SMT. In *ACC*, pages 1823–1829. IEEE, 2020.

[26] Manish Goyal and Parasara Sridhar Duggirala. On generating a variety of unsafe counterexamples for linear dynamical systems. *IFAC-PapersOnLine*, 51(16):139–144, 2018. ADHS.

[27] Manish Goyal and Parasara Sridhar Duggirala. Learning robustness of nonlinear systems using neural networks. *Design and Analysis of Robust Systems, DARS*, 2019.

[28] Manish Goyal and Parasara Sridhar Duggirala. Extracting counterexamples induced by safety violation in linear hybrid systems. *Automatica*, 117:109005, 2020.

[29] Manish Goyal and Parasara Sridhar Duggirala. Neuralexplorer: State space exploration of closed loop control systems using neural networks. In *ATVA*, volume 12302, pages 75–91. Springer, 2020.

[30] Ingo Gühring, Mones Raslan, and Gitta Kutyniok. Expressivity of deep neural networks. *arXiv preprint arXiv:2007.04759*, 2020.

[31] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *CAV*, pages 3–29, 2017.

[32] Zhenqi Huang and Sayan Mitra. Proofs from simulations and modular annotations. In *HSCC*, page 183–192. ACM, 2014.

[33] Radoslav Ivanov, Taylor Carpenter, James Weimer, Rajeev Alur, George Pappas, and Insup Lee. Verisig 2.0: Verification of neural network controllers using taylor model preconditioning. In *CAV*, pages 249–262. Springer, 2021.

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

[35] Taylor T Johnson, Diego Manzanas Lopez, Patrick Musau, Hoang-Dung Tran, Elena Botoeva, Francesco Leofante, Amir Maleki, Chelsea Sidrane, Jiameng Fan, and Chao Huang. Artificial intelligence and neural network control systems (ainncs) for continuous and hybrid systems plants. In *ARCH*, volume 74, pages 107–139. EasyChair, 2020.

[36] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2:255–299, 1990.

[37] Panagiotis Kyriakis, Jyotirmoy V. Deshmukh, and Paul Bogdan. Specification mining and robust design under uncertainty: A stochastic temporal logic approach. *ACM Trans. Embed. Comput. Syst.*, 18(5s), 2019.

[38] Sergey Levine, Peter Pastor, Alex Krizhevsky, Julian Ibarz, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, 37(4-5):421–436, 2018.

[39] F. L. Lewis, A. Yesildirak, and Suresh Jagannathan. *Neural Network Control of Robot Manipulators and Nonlinear Systems*. CRC Press 1998.

[40] Zichao Long, Yiping Lu, Xianzhong Ma, and Bin Dong. PDE-net: Learning PDEs from data. In *International Conference on Machine Learning*, volume 80, pages 3208–3216, Sweden, 2018. PMLR.

[41] Diego Manzanas Lopez, Patrick Musau, Hoang-Dung Tran, and Taylor T. Johnson. Verification of closed-loop systems with neural network controllers. In *ARCH*, volume 61, pages 201–210. EasyChair, 2019.

[42] Oded Maler, Dejan Nickovic, and Amir Pnueli. Checking temporal properties of discrete, timed and continuous behaviors. In *Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, pages 475–505. Springer, 2008.

[43] Yue Meng, Dawei Sun, Zeng Qiu, Md Tawhid Bin Waez, and Chuchu Fan. Learning density distribution of reachable states for autonomous systems, 2021.

[44] W. Thomas Miller, Richard S. Sutton, and Paul J. Werbos. *Neural Networks for Control*. The MIT Press, Cambridge, MA, USA, 01 1991.

[45] Kevin L Moore. *Iterative learning control for deterministic systems*. Springer Science & Business Media, London, 2012.

[46] Truong Nghiem, Sriram Sankaranarayanan, Georgios Fainekos, Franjo Ivancić, Aarti Gupta, and George J. Pappas. Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In *HSCC*, page 211–220, NY, USA, 2010. ACM.

[47] Dung Phan, Nicola Paoletti, Timothy Zhang, Radu Grosu, Scott A. Smolka, and Scott D. Stoller. Neural state classification for hybrid systems. In *International Workshop on Symbolic-Numeric Methods for Reasoning about CPS and IoT*, page 24–27, NY, USA, 2019. ACM.

[48] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Multistep neural networks for data-driven discovery of nonlinear dynamical systems. *arXiv*, 2018.

[49] Sriram Sankaranarayanan and Georgios Fainekos. Falsification of temporal properties of hybrid systems using the cross-entropy method. In *HSCC*, page 125–134. ACM, 2012.

[50] Dawei Sun and Sayan Mitra. Neureach: Learning reachability functions from simulations. In *TACAS*, pages 222–337. Springer, 2022.

[51] Xiaowu Sun, Haitham Khedr, and Yasser Shoukry. Formal verification of neural network controlled autonomous systems. In *HSCC*, page 147–156, New York, USA, 2019. ACM.

[52] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. Structural test coverage criteria for deep neural networks. *ACM Trans. Embed. Comput. Syst.*, 18(5s), 2019.

[53] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[54] Hoang-Dung Tran, Feiyang Cai, Manzanas Lopez Diego, Patrick Musau, Taylor T. Johnson, and Xenofon Koutsoukos. Safety verification of cyber-physical systems with reinforcement learning control. *ACM Trans. Embed. Comput. Syst.*, 18(5s), Oct 2019.

[55] Petra Vidnerová. RBF-Keras: an RBF Layer for Keras Library. https://github.com/PetraVidnerova/rbf_keras/, 2019.

[56] Weiming Xiang, Hoang-Dung Tran, Xiaodong Yang, and Taylor T. Johnson. Reachable set estimation for neural network control systems: A simulation-guided approach. *IEEE Transactions on Neural Networks and Learning Systems*, 32(5):1821–1830, 2021.

[57] Aditya Zutshi, Jyotirmoy V. Deshmukh, Sriram Sankaranarayanan, and James Kapinski. Multiple shooting, cegar-based falsification for hybrid systems. In *EMSOFT*. ACM, 2014.