NEURAL NETWORK VERIFICATION TOOLBOX

USER MANUAL V1.0

By

Dung Hoang Tran

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

Overview of NNV and its Features

I.1 Overview

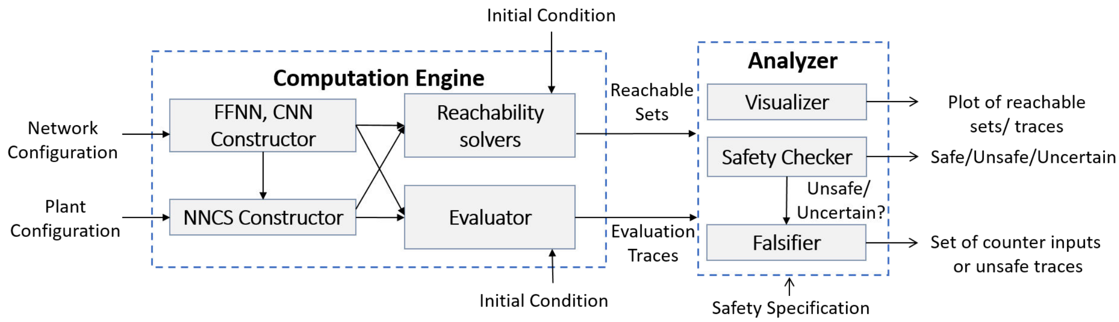

Figure I.1: An overview of NNV.

NNV[1] is an object-oriented toolbox which is built on top of the MPT toolbox Kvasnica et al. (2004) and CORA Althoff (2015) in Matlab. NNV makes use of the Neural Network Model Transformation Tool (nnmt [2]) which supports transforming neural network models from Keras and Tensorflow into Matlab using the Open Neural Network Exachange format, and the hybrid systems model transformation and translation tool (HyST) Bak et al. (2015) for plant configurations.

The NNV toolbox contains two main modules: a *computation engine* and an *analyzer*, shown in Figure I.1. The computation engine consists of four sub-components: 1) the *FFNN constructor*, 2) the *NNCS constructor*, 3) *the reachability solvers*, and 4) *the evaluator*. The FFNN constructor takes a network configuration file as an input and generates a FFNN object. The NNCS constructor takes a FFNN object and a plant configuration, which describes the dynamics of a system, as inputs and then creates an NNCS object. Depending on the application, either the FFNN (or NNCS) object will be passed to a reachability solver to compute the reachable set of the object (FFNN or NNCS) from a given initial set of states. Then, the obtained reachable set will be passed to the analyzer module. The analyzer module consists of three sub-components: 1) a *visualizer*, 2) a *safety checker*, and 3) a *falsifier*. The visualizer can be called to plot the obtained reachable set. Given a safety specification, the safety checker can reason about the safety of the FFNN or NNCS

[1]https://github.com/verivital/nnv
[2]https://github.com/verivital/nnmt

with respect to the specification. When an exact (sound and complete) reachability solver is used, such as the star-based solver, the safety checker can return either ”safe” or ”unsafe,” along with a set of counterexamples. When an over-approximate (sound) reachability solver is used, such as the zonotope-based scheme or the approximate star-based solvers, the safety checker can return either ”safe” or ”*uncertain*” (unknown). In this case, the falsifier automatically calls the evaluator to generate simulation traces to find a counterexample. If the falsifier can find a counterexample, then NNV returns unsafe. Otherwise, it returns unknown.

I.2 Features

NNV implements a set of reachability algorithms for sequential FFNNs and CNNs, It also implements reachability algorithms for NNCS with FFNN controllers. A summary of NNV's major features is given in Table I.1.

Feature	**Exact Analysis**	**Over-approximate Analysis**
Components	FFNN, CNN, NNCS	FFNN, CNN, NNCS
Plant dynamics (for NNCS)	Linear ODE	Linear ODE, Nonlinear ODE
Discrete/Continuous (for NNCS)	Discrete Time	Discrete Time, Continuous Time
Activation functions	ReLU, Satlin	ReLU, Satlin, Sigmoid, Tanh
CNN Layers	MaxPool, Conv, BN, AvgPool, FC	MaxPool, Conv, BN, AvgPool, FC
Reachability methods	Star, Polyhedron, ImageStar	Star, Zonotope, Abstract-domain, ImageStar
Reachable set/Flow-pipe Visualization	Yes	Yes
Parallel computing	Yes	Partially supported
Safety verification	Yes	Yes
Falsification	Yes	Yes
Robustness verification (for FFNN/CNN)	Yes	Yes
Counterexample generation	Yes	Yes

Table I.1: Overview of NNV's features. Each link refers to the relevant files/classes in NNV's codebase. BN, FC, AvgPool, Conv, MaxPool refer to batch normalization layers, fully-connected layers, average pooling layers, convolutional layers and max pooling layers respectively.

CHAPTER II

Installation

II.1 Operating System

Window 10, Mac, and Linux are supported. (CodeOcean described below is run on Linux).

II.2 Dependencies

Matlab 2018b or later (may work on earlier versions, but untested).

II.3 Installation Steps

- Clone or download the nnv toolbox from https://github.com/verivital/nnv. NNV depends on other tools (CORA, NNMT, HyST) to operate correctly, which are included as git submodules. As such, you must clone recursively. The following correctly clones the toolbox:
  git clone –recursive https://github.com/verivital/nnv.git

- Open matlab, then go to the directory where NNV exists on your machine and run the install.m script located at ../nnv/.

II.4 Execution without installation

NNV can be executed online without installing Matlab or other dependencies through CodeOcean, via the CodeOcean capsule DOI 10.24433/CO.1314285.v1: (https://doi.org/10.24433/CO.1314285.v1).

CHAPTER III

Verification of Feedforward Neural Networks (FFNN) Using NNV

III.1 Main steps

Using NNV for the verification of feedforward neural networks (FFNNs) consists of seven main steps:

- Constructing a FFNN object.
- Specifying a property of the network that we want to verify.
- Choosing a reachability analysis method
- Constructing an input set with which to verify the network.
- Choosing the number of cores utilized for computation.
- Verifying the network.
- Visualizing the results.

III.2 Constructing a FFNN

There are two ways to construct an FFNN object. The first is to manually construct the object layer-by-layer. The second option is to parse in a trained network from Matlab, Keras, Tensorflow or the ONNX format.

III.2.1 Manually Constructing an FFNN Object.

This is suitable when the users are familiar with all of the information about the network such as the weight matrices, bias vectors, and activation functions for each layer of the network. An FFNN object can be constructed by an array of layer objects. In the following example, we construct an FFNN with two layers with activation functions 'poslin' (ReLU) and 'pureline' (linear) respectively. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_ffnns_constructor.m.

Code 1: Construct manually an FFNN object

```
/* An example of manually create nnv FFNN object */

W1 = [1 -1; 0.5 2; -1 1]; % first layer weight matrix
b1 = [-1; 0.5; 0];        % first layer bias vector

W2 = [-2 1 1; 0.5 1 1];   % second layer weight matrix
b2 = [-0.5; -0.5];        % second layer bias vector

L1 = LayerS(W1, b1, 'poslin');  % 1st layer
L2 = LayerS(W2, b2, 'purelin'); % 2nd layer

F = FFNNS([L1 L2]); %  nnv feedforward neural network
```

Code 2: Result of the NNV Layer objects

```
L1 =
  LayerS with properties:
    W: [3x2 double] % weight matrix
    b: [3x1 double] % bias vector
    f: 'poslin'     % activation function
    N: 3            % number of neurons
L2 =
  LayerS with properties:
    W: [2x3 double] % weight matrix
    b: [2x1 double] % bias vector
    f: 'purelin'    % activation function
    N: 2            % number of neurons (or outputs)
```

Code 3: Result of the NNV FFNN object

```
F =

  FFNNS with properties:

              Name: 'net' % name of the network

            Layers: [1  2 LayerS]  % layer objects

                nL: 2  % number of layers

                nN: 5  % total number of neurons

                nI: 2  % number of inputs

                nO: 2  % number of outputs

       reachMethod: 'exact-star'

       reachOption: []

          numCores: 1

          inputSet: []

          reachSet: []

         outputSet: []

         reachTime: []

       numReachSet: []

    totalReachTime: 0

        numSamples: 2000

      unsafeRegion: []

        Operations: []
```

III.2.2 Automatically Constructing an FFNN Object Using a Matlab Network.

If the users train a network in Matlab and save the network parameters to a mat file, NNV can conveniently parse the trained network and automatically construct an equivalent FFNN object that can be used for verification. In the following example, we construct an FFNN object by parsing a toy example which is trained by General Motors' researchers using Matlab. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_ffnns_parse.m.

Code 4: Automatically constructing an FFNN object by parsing

```
/* An example of parsing a network trained in Matlab */

load Engine_Toy_Tansig_net.mat; % load the network

F = FFNNS.parse(net); % parse the network
----------------------------------------------------
Result:
----------------------------------------------------
F =
  FFNNS with properties:
              Name: 'net'
            Layers: [1x3 LayerS] % layer objects
                nL: 3    % number of layers
                nN: 21   % total number of neurons
                nI: 2    % number of inputs
                nO: 1    % number of outputs
       reachMethod: 'exact-star'
       reachOption: []
          numCores: 1
          inputSet: []
          reachSet: []
         outputSet: []
         reachTime: []
       numReachSet: []
    totalReachTime: 0
        numSamples: 2000
      unsafeRegion: []
        Operations: []
```

***IMPORTANT NOTE.** The network parsed by NNV and the originial matlab network are EQUIVALENT ONLY IF in the training process, the users do not set a pre-process function for the input, e.g., *mapminmax*. If a pre-process function is selected in the training process of a Matlab network, in prediction phase, one can test that with the same input, the output of the Matlab network and the NNV network are different. This is because, the Matlab network normalizes the input using the input process function before computing the prediction while NNV network does not. It should be emphasized that NNV network is constructed to compute the reachable sets of the network with a given input set. Normalization of the input set is required to be done by users.

III.2.3 Automatically Construct a FFNNS object using NNVMT

NNV can also construct a feedforward fully-connected neural network in various formats such as the ONNX format using the tool called nnvmt.

We support importing these networks from Keras, Tensorflow, Sherlock's format () and Reluplex's format (.nnet). For all of these formats, we support the common activation functions considered in the community: sigmoid, tanh, ReLU and linear. If imported from Keras, we also support a saturation linear function, with ranges 0 to 1 and -1 to 1, named satlin and satlins respectively (in MATLAB). The syntax for the function is as follows:

Code 5: Construct a FFNNS using NNVMT

```
fnn = load_nn(in1, in2, in3, in4, in5, opt);
```

Where:

- fnn is the output of the function, the neural network in NNV format (FFNNS)
- *in1 = python_path*: path to the python environment to use (where nnvmt requirements are installed)
- *in2 = nnvmt_path*: path to the nnmt folder installed
- *in3 = input_path*: path to the neural network file we want to transform/input to NNV
- *in4 = output_path*: path to save the transformed neural network into (as .mat file)

- *in5 = formatting*: format of the neural network to transform. Choose one of the following:
  - keras
  - tensorflow
  - onnx
  - sherlock
  - reluplex
- *opt*: this is an optional input. Depending on the format we are converting from, we will choose a different input here, but we will mostly use it to load neural networks trained in Keras, only when we have both an *.h5* and *.json* file for our NN. The .h5 file will be the 3^{rd} input (input_path), and the .json file will be the 6^{th} input (opt).

Some examples of using NNVMT to create a FFNNS are

Code 6: Example from Reluplex

```
fnn = load_nn('../python','nnv/engine/nnmt', 'name_of_nn.nnet',
'../networks','reluplex');
```

Code 7: Example from Sherlock

```
fnn = load_nn('../python','nnv/engine/nnmt', 'name_of_nn',
'../networks','sherlock');
```

Code 8: Example from Keras (1)

```
fnn = load_nn('../python','nnv/engine/nnmt', 'name_of_nn.h5',
'../networks','keras');
```

Code 9: Example from Keras (2)

```
fnn = load_nn('../python','nnv/engine/nnmt', 'name_of_nn.h5',
'../networks','keras','name_of_nn.json');
```

Code 10: Example from Tensorflow

```
fnn = load_nn('../python','nnv/engine/nnmt', 'name_of_nn'*,
'../networks','tensorflow');
```

*Note: In the case of Tensorflow, based on the common practice of saving both the checkpoint and the .meta file in the same folder, in3 (input_path) will point to the folder where these two files are located.

Code 11: Example from ONNX

```
fnn = load_nn('../python','nnv/engine/nnmt', 'name_of_nn.onnx',
'../networks','onnx');
```

As an additional feature, after transforming any network into name_of_nn.mat file, you can load this file using load_nn.m as well. Simply use the function by leaving all of the inputs blank and specifying the path to the .mat file in the opt position. Example:

Code 12: Construct a FFNNS after transforming to .mat

```
fnn = load_nn('','','','','','name_of_nn.mat');
```

For more details, please visit this tool at: https://github.com/verivital/nnvmt.

III.3 Specifying a property of an FFNN

After constructing a FFNN network, the users need to specify the property that they are considering about the network that they want to verify. The property is a linear predicate over the outputs of the network which is defined in the form of $P \triangleq Gy \leq g$, where y is the output vector of the network. Let P be an unsafe region, if the reachable sets of the network reach the unsafe region, the network is unsafe, otherwise, it is safe. In NNV, we use a *HalfSpace* object to represent a property. In the following example, we specify an unsafe region for the network constructed in section III.2.1. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_ffnns_specify_property.m.

Code 13: Specify a property for FFNN

```
/* An example of specifying a property of FFNN */
% unsafe region: y1 >= 5 (the network has two outputs)
G = [-1 0]; % condition matrix
g = -5;     % condition vector
U = HalfSpace(G, g); % unsafe region object
-----------------------------------------------------
Result
-----------------------------------------------------
U =
  HalfSpace with properties:
      G: [-1 0] % condition matrix
      g: -5     % condition vector
    dim: 2      % dimension of the space
```

III.4 Choosing a Reachability Method

To verify whether a network violates a (safety) property or not, NNV computes the reachable set of the network corresponding to a specific input set. NNV supports different reachability schemes for FFNNs including *"exact-star"*, *"approx-star"*, *"exact-polyhedron"*, *"approx-zono"*, and *"abs-dom"* as depicted in Table I.1. The following example chooses *"exact-star"* as the reachability method for the verification of the network constructed in section III.2.1. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_ffnns_choose_reach_method.m.

Code 14: Choosing reachability method for FFNN

```
/* An example of choosing reachability method for FFNN */
reachMethod = 'exact-star';
-----------------------------------------------------------
Result
-----------------------------------------------------------
reachMethod =
    'exact-star'
```

III.5 Constructing an Input Set for an FFNN

The input set to the network could be a *star set*, a *zonotope* or a *polyhedron*, depending on the reachability method used for verification.

III.5.1 Constructing a Star Input Set

A star input set is required when we use the *"exact-star"*, *"approx-star"*, and *"abs-dom"* reachability methods. *A star set* (or simply star) Θ is a tuple $\langle c, V, P \rangle$ where $c \in \mathbb{R}^n$ is the center, $V = \{v_1, v_2, \cdots, v_m\}$ is a set of m vectors in $\mathbb{R}^n$ called basis vectors, and $P : \mathbb{R}^m \rightarrow \{\top, \bot\}$ is a predicate. The basis vectors are arranged to form the star's $n \times m$ basis matrix. In NNV, we restrict the predicates to be a conjunction of linear constraints, $P(\alpha) \triangleq C\alpha \leq d$ where, for p linear constraints, $C \in \mathbb{R}^{p \times m}$, α is the vector of m-variables, i.e., $\alpha = [\alpha_1, \cdots, \alpha_m]^T$, and $d \in \mathbb{R}^{p \times 1}$. A star is an empty set if and only if $P(\alpha)$ is empty. The set of states represented by the star is given as:

$$[\![\Theta]\!] = \{x \mid x = c + \Sigma_{i=1}^{m}(\alpha_i v_i) \text{ such that } P(\alpha_1, \cdots, \alpha_m) = \top\}. \tag{III.1}$$

To construct a star set, we use two common methods. The first constructs a star set when all information of the set is known, i.e., we have $\{c, v_1, \ldots, v_m, C, d\}$. In NNV, we combine the center vector c and the basis vectors v_j into a single basis matrix $V = [c\ v_1\ \ldots\ v_m]$. The second method constructs a star set from the ranges of all individual inputs. The following example constructs a star set using different approaches. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_ffnns_choose_construct_star.m.

Code 15: Construct a star set when all information is known

```
/* An example of constructing a star set */
c1 = [1; -1]; % center vector
v1 = [1; 0];  % basis vector 1
v2 = [0; 0.5]; % basis vector 2
V = [c1 v1 v2]; % basis matrix
% predicate constraint: P = C*[a] <= d
% -1<= a1 <= 1, 0 <= a2 <= 1, a1 + a2 <= 1
C = [1 0; -1 0; 0 1; 0 -1; 1 1]; % constraint matrix
d = [1; 1; 1; 0; 1];             % constraint vector
I1 = Star(V, C, d); % star input set
----------------------------------------------------
Result
----------------------------------------------------
I1 =
  Star with properties:
               V: [2x3 double] % basis matrix
               C: [5x2 double] % constraint matrix
               d: [5x1 double] % constraint vector
             dim: 2  % dimension of star
            nVar: 2  % number of predicates variables
    predicate_lb: [2x1 double] % lower bound of predicate vars
    predicate_ub: [2x1 double] % upper bound of predicate vars
        state_lb: []  % lower bound state vector
        state_ub: []  % upper bound state vector
```

Code 16: Construct a star set from input ranges

```
/* An example of constructing a star set from input ranges */
% -2 <= x1 <= 2
% 0 <= x2 <= 1
lb = [-2; 0]; % lower bound vector
ub = [2; 1]; % upper bound vector

I2 = Star(lb, ub); % star input set
------------------------------------------------------------
Result
------------------------------------------------------------
I2 =
  Star with properties:
               V: [2x3 double] % basis matrix
               C: [4x2 double] % constraint matrix
               d: [4x1 double] % constraint vector
             dim: 2
% dimension of star
            nVar: 2  % number of predicates variables
    predicate_lb: [2x1 double] % lower bound of predicate vars
    predicate_ub: [2x1 double] % upper bound of predicate vars
        state_lb: [2x1 double] % lower bound state vector
        state_ub: [2x1 double] % upper bound state vector
```

III.5.2 Constructing a Polyhedron Input Set

A polyhedron input is required when we use the "exact-polyhedron" reachability method for verification. In NNV, we require the polyhedron input set to be a bounded set. NNV uses the MPT toolbox Kvasnica et al. (2004) to construct, manipulate and visualize a polyhedron. A polyhedron is defined as,

$$P = \{x \mid Ax \leq b,\ A_e x = b_e\} \tag{III.2}$$

There are two common ways to construct a polyhedron. The first one uses the polyhedron related matrices A, B, A_e, and B_e. The second one constructs a polyhedron from the ranges of the states. In this case, a polyhedron is a hyper-rectangle. The following examples construct a polyhedron using different approaches. The code for these example is available at https://github.com/verivital/nnv/code/example/Manual/example_ffnns_choose_construct_polyhedron.m.

Code 17: Construct a polyhedron input set from input ranges

```
/* An example of constructing a polyhedron input set */
lb = [-1; 1]; % lower bound vector
ub = [1; 2];  % upper bound vector
% polyhedron from input ranges
I1 = Polyhedron('lb', lb, 'ub', ub);
-----------------------------------------------------
Result
-----------------------------------------------------
I1
Polyhedron in R^2 with representations:
  H-rep (redundant)   : Inequalities   4 | Equalities   0
  V-rep               : Unknown (call computeVRep() to compute)
Functions : none
```

Code 18: Construct a polyhedron input set from matrices

```
/* An example of constructing a polyhedron input set */
A = [2 1; 1 0; -1 0; 0 1; 0 -1; 1 1]; % inequality matrix
b = [2; 1; 1; 0; 1; 1]; % inequality vector
I2 = Polyhedron('A', A, 'b', b);% polyhedron without equalities
Ae = [2 3]; % equality matrix
be = 1.5;   % equality vector
% polyhedron with one equality
I3 = Polyhedron('A', A, 'b', b, 'Ae', Ae, 'be', be);
---------------------------------------------------------------
Results
--------------------------------------------------------------
I2
Polyhedron in R^2 with representations:
  H-rep (redundant)   : Inequalities   6 | Equalities   0
  V-rep               : Unknown (call computeVRep() to compute)
Functions : none
I3
Polyhedron in R^2 with representations:
  H-rep (redundant)   : Inequalities   6 | Equalities   1
  V-rep               : Unknown (call computeVRep() to compute)
Functions : none
```

III.5.3 Constructing a Zonotope Input Set

A zonotope input set is required when we use "approx-zono" reachability method for verification. A zonotope has a similar structure as a star except that all predicate variables are in the ranges of $[-1,\ 1]$. Mathematically, a zonotope is defined as:

$$Z = \{x \mid x = c + \Sigma_i^m \alpha_i \times v_i\}, \tag{III.3}$$

where c is the center vector, v_i is a *generator* (we just call a basis vector) and α_i is a predicate variable of the range $[-1,\ 1]$.

In NNV, we can construct a zonotope if we know the center vector c and the basis matrix $V = [v_1\ v_2\ \dots\ v_m]$. We can also construct a zonotope if we know the ranges of all individual inputs. The following example constructs a zonotope using different approaches. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_ffnns_choose_construct_zonotope.m.

Code 19: Construct a zonotope input set

```
/* An example of constructing zonotope input set */
c = [-1; 0]; % center vector
v1 = [2; 1]; % 1st basis vector
v2 = [1; 1]; % 2nd basis vector
v3 = [-1; 1]; % 3rd basis vector
V = [v1 v2 v3]; % basis matrix
% zonotope from center vector and basis matrix
I1 = Zono(c, V);
lb = [-1; 1]; % lower bound vector
ub = [1; 2];  % upper bound vector
% zonotope from input ranges
I2 = Box(lb, ub); % a box object
I2 = I2.toZono;   % transform to zonotope
---------------------------------------------------
Results
---------------------------------------------------
I1 =
  Zono with properties:
      c: [2x1 double] % center vector
      V: [2x3 double] % basis matrix
    dim: 2            % dimenison
I2 =
  Zono with properties:
      c: [2x1 double] % center vector
      V: [2x2 double] % basis matrix
    dim: 2            % dimension
```

III.6 Choosing the Number of Cores Utilized for Computation

The number of cores utilized for computation is only of concern when using the exact reachability methods (*"exact-star"* and *"exact-polyhedron"*). The over-approximate reachability methods (*"approx-star"*, *"approx-zono"*, and *"abs-dom"*) use one core for the computation. For FFNNs with pieecwise linear activiation functions such as ReLU (poslin) and Saturation (satlin), NNV can compute the exact reachable set of the network using the *"exact-star"* and *"exact-polyhedron"* reachability schemes. The *"exact-star"* method is faster and more scalable than the *"exact-polyhedron"* method.

NNV computes the reachable set of the network layer-by-layer, i.e., the output of the current layer is the input for the next layer. In the exact analysis, a single input set can split into multiple output sets after one layer. Therefore, to speed up the computation for the exact analysis, NNV uses the *"parfor"* option in the Matlab Parallel Computing ToolBox, i.e., a layer computation can independently handle multiple input sets at the same time using multiple cores. Therefore, the users need to choose the number of cores they want to use for the computation which depend on the configuration of their machines. The following example simply chooses 4 cores for the computation.

Code 20: Choose number of cores for computation

```
/* An example of choosing a number of cores */
numCores = 4; % use 4 cores for computation
```

III.7 Verifying an FFNN

The steps taken so far have been:

- Constructing an FFNN object.
- Specifying a property about the network that we want to verify.
- Choosing a reachability analysis method
- Constructing an input set with which we want to verify the network.
- Choosing the number of cores utilized for computation.

There are two options for verifying an FFNN that users can select. The first option is to call the *reach* method on the FFNN object to compute the output sets of the network. Then, users can verify if all the output sets satisfy the property. The first option gives a deeper understanding how NNV verifies a network as we divide the reachability problem and the verification problem into two separate steps. The second option is an automatic combination of these two steps by calling *verify* on the FFNN object.

III.7.1 Manually Verifying an FFNN

III.7.1.1 Computing Output Reachable Sets

To compute the output reachable sets of an FFNN, we use the *reach* method on the network object. In the following example, we use different reachability methods to compute the output reachable sets of the network constructed in section III.2.1. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_ffnns_compute_reachSet.m.

Code 21: Compute reachable sets of an FFNN

```
/* An example of computing reachable sets of an FFNN */
/* construct an NNV network
W1 = [1 -1; 0.5 2; -1 1];
b1 = [-1; 0.5; 0];
W2 = [-2 1 1; 0.5 1 1];
b2 = [-0.5; -0.5];
L1 = LayerS(W1, b1, 'poslin');
L2 = LayerS(W2, b2, 'purelin');
F = FFNNS([L1 L2]); % construct an NNV FFNN

/* choose the number of cores
numCores = 2;

/* construct input set
lb = [-1; -2]; % lower bound vector
ub = [1; 0]; % upper bound vector
I = Star(lb, ub); % star input set
I_Poly = Polyhedron('lb', lb, 'ub', ub); % polyhedron input set
B = Box(lb, ub); % a box input set
I_Zono = B.toZono; % convert to a zonotope

/* compute the reachable sets with a selected method
[R1, t1] = F.reach(I, 'exact-star', numCores);
[R2, t2] = F.reach(I_Poly, 'exact-polyhedron', numCores);
[R3, t3] = F.reach(I, 'approx-star');
[R4, t4] = F.reach(I_Zono, 'approx-zono');
[R5, t5] = F.reach(I, 'abs-dom');
```

Code 22: Reachable sets and computation time results

```
R1 = 1x6 Star array with properties:

    V

    C

    d

    dim

    nVar

    predicate_lb

    predicate_ub

    state_lb

    state_ub

t1 =  0.1347

R2 = Array of 6 polyhedra.

t2 =    0.3003

R3 = Star with properties:

               V: [2x6 double]

               C: [13x5 double]

               d: [13x1 double]

             dim: 2

            nVar: 5

    predicate_lb: [5x1 double]

    predicate_ub: [5x1 double]

        state_lb: []

        state_ub: []

t3 =  0.0134
```

Code 23: Reachable sets and computation time results (cont.)

```
R4 = Zono with properties:
      c: [2x1 double]
      V: [2x5 double]
    dim: 2
t4 =  0.0069
R5 =  Star with properties:
               V: [2x6 double]
               C: [10x5 double]
               d: [10x1 double]
             dim: 2
            nVar: 5
    predicate_lb: [5x1 double]
    predicate_ub: [5x1 double]
        state_lb: []
        state_ub: []
t5 =  0.0064
```

We can see that the exact reachable set of the network contains 6 star sets or 6 polyhedra. The *"exact-star"* method is faster than the *"exact-polyhedron"* method ($t1 = 0.1347 < t2 = 0.3003$). The over-approximate reachability method produces a single output set which can be a star or a zonotope depending on which method is used.

III.7.1.2 Verifying Output Reachable Sets

With the output reachable sets computed previously, we can verify whether or not they violate a property of the network. Assume that we want to verify the following property:

$$P = \{y \in R^2 | \, y_1 \geq 1.5\}. \tag{III.4}$$

To prove that the network satisfies (SAT) or does not satisfy (UNSAT) the property, we check the intersection between the reachable sets with the property given in the following example. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_ffnns_verify_reachSet.m.

Code 24: Verify output reachable sets

```
/* An example of verifying output reachable sets */
...
P = HalfSpace([-1 0], -1.5); % P: y1 >= 1.5
P_Poly = Polyhedron('A', P.G, 'b', P.g);
rs1 = cell(1, length(R1));
rs2 = cell(1, length(R2));
for i=1:length(R1)
    M = R1(i).intersectHalfSpace(P.G, P.g); % verifying R1
    if isempty(M)
        rs1{i} = 'UNSAT';
    else
        rs1{i} = 'SAT';
    end
end
for i=1:length(R2)
    M = R2(i).intersect(P_Poly); % verifying R2
    if M.isEmptySet
        rs2{i} = 'UNSAT';
    else
        rs2{i} = 'SAT';
    end
----------------------------------------------------------
Results
----------------------------------------------------------
rs1 = rs2 = 1x6 cell array
{'UNSAT'}{'UNSAT'}{'UNSAT'}{'UNSAT'}{'UNSAT'}{'UNSAT'}
```

Code 25: Verify output reachable sets (cont.)

```
/* An example of verifying output reachable sets */
...
M = R3.intersectHalfSpace(P.G, P.g); % verify R3
if isempty(M)
    rs3 = 'UNSAT';
else
    rs3 = 'SAT';
end
M = R4.intersectHalfSpace(P.G, P.g); % verify R4
if isempty(M)
    rs4 = 'UNSAT';
else
    rs4 = 'SAT';
end
M = R5.intersectHalfSpace(P.G, P.g); % verify R5
if isempty(M)
    rs5 = 'UNSAT';
else
    rs5 = 'SAT';
end
--------------------------------------------------------
Results
--------------------------------------------------------
rs3 = 'UNSAT';
rs4 = 'SAT';
rs5 = 'SAT';
```

One can see that, when using the exact reachability methods, the output reachable sets do not *reach* the property P. Therefore NNV returns UNSAT results for all reachable sets. When over-approximate reachability methods are used, only the *"approx-star"* method returns UNSAT while the *"approx-zono"* and the *"abs-dom"* methods return SAT. This is because the reachable sets obtained by the *"approx-zono"* and the *"abs-dom"* methods are conservative and reach the property. Users can observe the conservativeness of different over-approximate reachability approaches by visualizing the reachable sets of the network. This visualization is addressed in the next section. The conservativeness of the obtained reachable sets is very important in proving the safety of a network. When the reachable sets of the network reach an unsafe region, we say the network is unsafe, otherwise, it is safe. Due to the conservativeness of the reachable set computation, it is possible that a network is safe but the obtained over-approximate reachable set reaches the unsafe region. In this case, we cannot prove the safety of the network using the over-approximate reachability techniques.

III.7.2 Automatically Verifying an FFNN

After specifying a property, we can automatically verify an FFNN by using the *verify* method. In this method, we specify the property as an unsafe property. The verification result may be *"safe"*, *"unsafe"*, or *"unknown"*. We do not use *"exact-polyhedron"* methods due to their low scalability. To start, the *verify* method runs some random simulations by randomly sampling the input set to see if the unsafe region is reached. If not, it performs reachability analysis to compute the reachable sets and then proves the safety of the network. Therefore, besides the *input set*, *unsafe property*, and *number of core* parameters, we have one more parameter called the *number of samples*, that is used for randomly generating simulations. If the users set this parameter to zero, then the *verify* method neglects randomly generating simulations.

Code 26: Automatically Verifying an FFNN

```
/* An example of automatically verifying an FFNN */
/* construct an NNV network
W1 = [1 -1; 0.5 2; -1 1];
b1 = [-1; 0.5; 0];
W2 = [-2 1 1; 0.5 1 1];
b2 = [-0.5; -0.5];
L1 = LayerS(W1, b1, 'poslin');
L2 = LayerS(W2, b2, 'purelin');
F = FFNNS([L1 L2]); % construct an NNV FFNN
/* construct input set
lb = [-1; -2]; % lower bound vector
ub = [1; 0]; % upper bound vector
I = Star(lb, ub); % star input set
B = Box(lb, ub); % a box input set
I_Zono = B.toZono; % convert to a zonotope
/* Properties
P = HalfSpace([-1 0], -1.5); % P: y1 >= 1.5
/* verify the network
nC = 1; % number of cores
nS = 100; % number of samples
[safe1, t1, cE1] = F.verify(I, P, 'exact-star', nC, nS);
[safe2, t2, cE2] = F.verify(I, P, 'approx-star', nC, nS);
[safe3, t3, cE3] = F.verify(I_Zono, P, 'approx-zono', nC, nS);
[safe4, t4, cE4] = F.verify(I, P, 'abs-dom', nC, nS);
```

Code 27: Results

```
safe1 =  1;                                  % safe
t1    =  0.3892;    % verification time
cE1   = [];         % counter examples
safe2 =  1;                                  % safe
t2    =  0.2907;    % verification time
cE2   = [];         % counter examples
safe3 =  2;                                  % unknown
t3    =  0.3300;    % verification time
cE3   = [];         % counter examples
safe4 =  2;                                  % unknown
t4    =  0.3213;    % verification time
cE4   = [];         % counter examples
```

We can see that the *"exact-star"* and *"approx-star"* methods can prove the safety of a network. The reachable sets obtained by these methods do not reach the unsafe region, i.e., property P. Since the network is safe, there are no counterexamples in this case. A counterexample is an input that make the network unsafe, i.e., the output of the network corresponding to the counter input relies in the unsafe region. When we use the *"approx-zono"* and *"abs-dom"* methods, we cannot prove the safety of the network. The reachable sets obtained by these methods reach the unsafe region. However, we do not know whether the exact reachable set of the network reaches the unsafe region or because of the conservativeness of the over-approximation reachable sets. We can conclude that the *"approx-star"* method is less conservative than the *"approx-zono"* and *"abs-dom"* methods.

If we change the property P into $y_1 \geq 0.4$, we have a new verification results as follows.

Code 28: Verify automatically an FFNN

```
/* An example of automatically verifying an FFNN */
...
/* Properties
P = HalfSpace([-1 0], -0.4); % P: y1 >= 1.5
/* verify the network
nC = 1; % number of cores
nS = 100; % number of samples
[safe1, t1, cE1] = F.verify(I, P, 'exact-star', nC, nS);
[safe2, t2, cE2] = F.verify(I, P, 'approx-star', nC, nS);
[safe3, t3, cE3] = F.verify(I_Zono, P, 'approx-zono', nC, nS);
[safe4, t4, cE4] = F.verify(I, P, 'abs-dom', nC, nS);
---------------------------------------------------------------
Results
---------------------------------------------------------------
safe1 =  0;         % unsafe
t1    =  0.3079;    % verification time
cE1 = 1x4 Star array; % counter examples
safe2 =  2;         % unknown
t2    =  0.2581;    % verification time
cE2   = [];         % counter examples
safe3 =  2        ; % unknown
t3    =  0.2478;    % verification time
cE3   = [];         % counter examples
safe4 =  2;         % unknown
t4    =  0.2435;    % verification time
cE4   = [];         % counter examples
```

We can see that the *"exact-star"* can prove that the network is unsafe. Additionally, it can compute all subsets of the input set that cause the network to be unsafe. The counterexamples are an array of 4 star sets which are the subsets of the input set. This is a unique feature of NNV. The user can increase the number of samples to see how it affects the results. When the number of samples increases, we can find counterexamples just by using random simulations. However, this approach is not efficient in general. We are working on better approaches for falsification of neural networks in the future.

III.8 Visualizing the results

Using NNV, users can intuitively observe the verification results by plotting the output reachable sets of the network and the unsafe region. After executing the *"verify"* method, if there are no counterexamples found by simulation, NNV performs reachability analysis to prove the safety of the network. The output reachable sets of the network are stored in the *FFNN.outputSet* property which can be visualized in some specific subspace using the *"visualize"* method in the FFNN object. This *visualize* method plots the mapped reachable sets of the output set in 2-D or 3-D space. The input to this method is a *mapping matrix* G and a *offset vector* g. Mathematically, if y is the output of the network, the *visual* method plots the set of $y' = G \times y + g$. Note that, because we can only visualize the reachable sets in 2-D or 3-D space, the *maximum allowable number of rows* of the *mapping matrix* and the *offset vector* is 3. If users do not use the offset vector, they can simply set it as an empty array or a zero vector.

The following example visualizes the reachable sets and the unsafe region of the network used in previous example. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_ffnns_visualize.m.

Code 29: Visualizing Verification results

```
/* An example of visualizing verification results of an FFNN */
...
map_mat = eye(2); map_vec = []; % mapping matrix & vector
P_poly = Polyhedron('A', P.G, 'b', P.g); % polyhedron obj
subplot(2, 2, 1);
[safe1, t1, cE1] = F.verify(I, P, 'exact-star', nC, nS);
F.visualize(map_mat, map_vec); % plot y1 y2
hold on;
plot(P_poly); % plot unsafe region
title('exact-star', 'FontSize', 13);
subplot(2,2,2);
[safe2, t2, cE2] = F.verify(I, P, 'approx-star', nC, nS);
F.visualize(map_mat, map_vec); % plot y1 y2
hold on;
plot(P_poly); % plot unsafe region
title('approx-star', 'FontSize', 13);
subplot(2,2,3);
[safe3, t3, cE3] = F.verify(I_Zono, P, 'approx-zono', nC, nS);
F.visualize(map_mat, map_vec); % plot y1 y2
hold on;
plot(P_poly); % plot unsafe region
title('approx-zono', 'FontSize', 13);
subplot(2, 2, 4);
[safe4, t4, cE4] = F.verify(I, P, 'abs-dom', nC, nS);
F.visualize(map_mat, map_vec); % plot y1 y2
hold on;
plot(P_poly); % plot unsafe region
title('abs-dom', 'FontSize', 13);
```

The visualization of the verification results is depicted in Figure III.1. From the figure, one can see that the *"approx-star"* method produces a smaller reachable set than the *"approx-zono"* and the *"abs-dom"* methods. The reachable sets of all methods reaches the unsafe region $P : y_1 \geq 0.4$ (the right unbounded region of the vertical black line). However, only the *"exact-star"* can prove the network is unsafe while the others cannot because of the over-approximation errors of the reachable sets. (We do not know if the unsafe region is reached because of the actual reachable sets, or the over-approximation error).

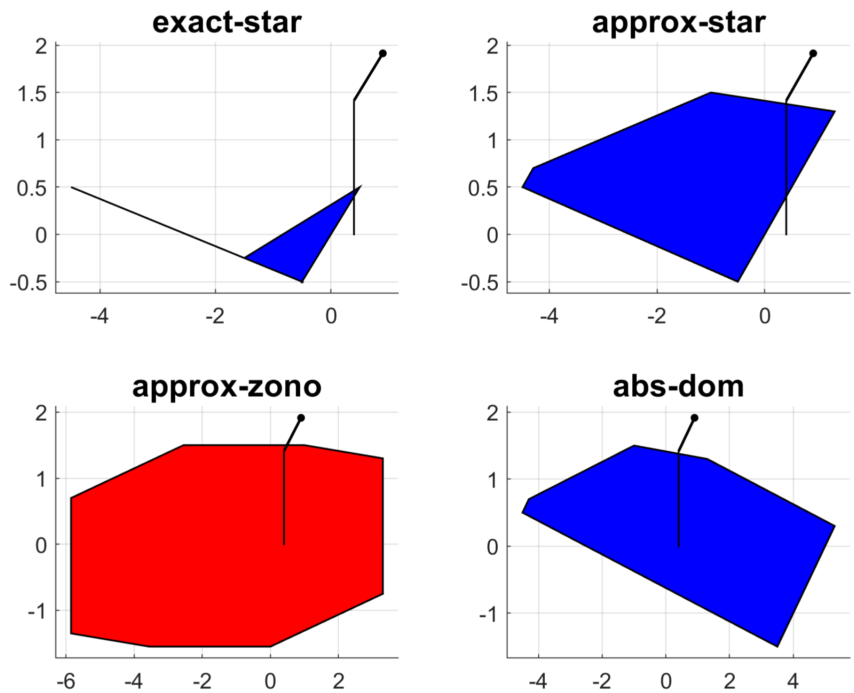

Figure III.1: A visualization of the verification results of an FFNN. The reachable sets of all methods reach the unsafe region $P : y_1 \geq 0.4$ (the right unbounded region of the vertical black line). However, only the *"exact-star"* can prove the network is unsafe while the others cannot because of the over-approximation errors of the reachable sets. (We do not know if the unsafe region is reached because of the actual reachable sets, or the over-approximation error).

CHAPTER IV

Verification of neural network control systems (NNCS) using NNV

IV.1 NNCS architecture

NNV supports verification of closed loop control systems with an ***FFNN controller with piecewise linear activation functions***. The architecture of such system is depicted in Figure IV.1. The plant model in the system can be continuous or discrete, linear or nonlinear.

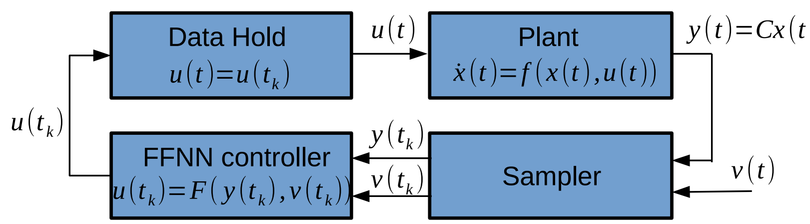

Figure IV.1: An architecture of NNCS supported in NNV.

IV.2 Main steps

The Verification of a neural network control system (NNCS) consists of seven main steps:

- Constructing an NNCS object.
- Specifying a property of the system that we want to verify.
- Choosing a reachability analysis method
- Constructing an initial set of states of the system.
- Choosing the number of cores utilized for computation.
- Verifying the system.
- Visualizing the results.

IV.3 Constructing an NNCS

IV.3.1 Four Types of NNCS

Depending on the plant model, NNV provides different classes of NNCS including:

1. The *LinearNNCS* class for NNCS with continuous linear plant models.

2. The *DLinearNNCS* class for NNCS with discrete linear plant models.

3. The *NonlinearNNCS* for NNCS with continuous nonlinear plant models.

4. The *DNonlinearNNCS* class for NNCS with discrete nonlinear plant models.

IV.3.2 Constructing a continuous linear NNCS

A continuous linear NNCS object is constructed using a FFNN controller object and a continuous linear plant model object.

IV.3.2.1 Constructing an FFNN controller object

This is a construction of an FFNN object. Please refer to section III.2 for details on how this object is constructed.

IV.3.2.2 Constructing a continuous linear plant object

A continuous linear plant is described by the following equation.

$$\dot{x}(t) = Ax(t) + Bu(t), y(t) = Cx(t) + Du(t). \tag{IV.1}$$

A continuous linear plant object is constructed using the *LinearODE* class in NNV. The inputs to the constructor of the *LinearODE* class are:

1. The system matrices A, B, C and D.

2. The *control period*, i.e., the plant takes control inputs at every *control period* seconds.

3. The *number of reachability steps* in one control period.

We note that in the verification of NNCS, we only consider the case of $y(t) = Cx(t)$. Therefore, the matrix D is set to empty. The following example constructs a continuous linear plant object. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_nncs_construct_linearODE.m.

Code 30: Constructing a Continuous Linear Plant Object

```
/* An example of constructing a continuous linear plant model */
A = [0 1;-5 -2]; % system matrix
B = [0;3];       % control matrix
C = [0 1];       % output feedback matrix
D = [];          % output control matrix
Tc = 0.1; % control period
Nr = 20; % number of reachability steps in one control period
sys = LinearODE(A, B, C, D, Tc, Nr); % plant object
---------------------------------------------------------------
Result
---------------------------------------------------------------
sys =
  LinearODE with properties:
                A: [2x2 double]
                B: [2x1 double]
                C: [0 1]
                D: []
              dim: 2
               nI: 1
               nO: 1
    controlPeriod: 0.1000
      numReachSteps: 20
```

IV.3.2.3 Constructing a Continuous Linear NNCS Object

After constructing an FFNN controller and a continuous linear plant object. A linear NNCS object can be constructed by feeding the FFNN controller object and the linear plant object into the constructor of the *LinearNNCS* class. The following example constructs an NNV continuous linear NNCS object. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_nncs_construct_linearNNCS.m.

Code 31: Constructing a Continuous Linear NNCS Object

```
/* An example of constructing a continuous linear NNCS object */
/* construct an FFNN controller
L1 = LayerS([2; 1], [0.5; -1], 'poslin');
L2 = LayerS([1 -1], 0.2, 'purelin');
F  = FFNNS([L1 L2]);


/* construct a plant model
A = [0 1;-5 -2]; % system matrix
B = [0;3];       % control matrix
C = [0 1];       % output feedback matrix
D = [];          % output control matrix
Tc = 0.1; % control period
Nr = 20; % number of reachability steps in one control period
sys = LinearODE(A, B, C, D, Tc, Nr); % plant object
/* construct a linear NNCS object
ncs = LinearNNCS(F, sys);
```

Code 32: Construction Results

```
ncs =

  LinearNNCS with properties:

                  controller: [1x1 FFNNS]
                       plant: [1x1 LinearODE]
                          nO: 1 % number of outputs
                          nI: 1 % number of inputs
                      nI_ref: 0 % ** unused
                       nI_fb: 1 % number of feedbacks
                      method: 'exact-star'
            plantReachMethod: 'direct'
                  transPlant: [1x1 LinearODE]
               plantReachSet: {}
    plantIntermediateReachSet: {}
           plantNumOfSimSteps: 20
               controlPeriod: 0.1000
           controllerReachSet: {}
                    numCores: 1
                       ref_I: []
                    init_set: []
                   reachTime: 0
                   simTraces: {}
               controlTraces: {}
               falsifyTraces: {}
                 falsifyTime: 0
```

IV.3.3 Constructing A Discrete Linear NNCS

Constructing a discrete linear NNCS object is similar to constructing a continuous linear NNCS object. The only difference is that we use the *DLinearODE* class to construct the discrete linear plant model.

IV.3.3.1 Constructing an FFNN controller object

This is a construction of an FFNN object. Please refer to section III.2 for details on how this object is constructed.

IV.3.3.2 Constructing a Discrete Linear Plant Object

A discrete linear plant model is defined as follows.

$$x[k+1] = Ax[k] + Bu[k], \; y[k] = Cx[k] + Du[k]. \tag{IV.2}$$

We note that in the verification of NNCS, we only consider the case of $y[k] = Cx[k]$. Therefore, the matrix D is set to empty. The following example constructs a discrete linear plant object. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_nncs_construct_dlinearODE.m.

Code 33: Constructing a Discrete Linear Plant Object

```
/* An example of constructing a discrete linear plant model */
A = [0 1;-5 -2]; % system matrix
B = [0;3];       % control matrix
C = [0 1];       % output feedback matrix
D = [];          % output control matrix
Ts = 0.1;        % sampling time
sys = DLinearODE(A, B, C, D, Ts); % plant object
--------------------------------------------------------------
Result
--------------------------------------------------------------
sys =
  DLinearODE with properties:
      A: [2x2 double] % system matrix
      B: [2x1 double] % control matrix
      C: [0 1] % output feedback matrix
      D: [] % output control matrix
     nI: 1 % number of inputs
     nO: 1 % number of outputs
    dim: 2 % system dimension
     Ts: 0.1000 % sampling time
```

IV.3.3.3 Constructing A Discrete Linear NNCS Object

After constructing an FFNN controller and a discrete linear plant object, a discrete linear NNCS object can be constructed by feeding the FFNN controller object and the discrete linear plant object into the constructor of the *DLinearNNCS* class. The following example constructs a discrete linear NNCS object. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_nncs_construct_dlinearNNCS.m.

Code 34: Constructing a Discrete Linear NNCS

```
/* An example of constructing an discrete linear NNCS object */
/* construct a FFNN controller
L1 = LayerS([2; 1], [0.5; -1], 'poslin');
L2 = LayerS([1 -1], 0.2, 'purelin');
F  = FFNNS([L1 L2]);

/* construct a plant model
A = [0 1;-5 -2]; % system matrix
B = [0;3];       % control matrix
C = [0 1];       % output feedback matrix
D = [];          % output control matrix
Ts = 0.1;        % sampling time
sys = DLinearODE(A, B, C, D, Ts); % plant object

/* construct a linear NNCS object
ncs = DLinearNNCS(F, sys);
```

Code 35: Construction results

```
ncs =

  DLinearNNCS with properties:

             controller: [1x1 FFNNS]
                  plant: [1x1 DLinearODE]
                     nO: 1 % number of output
                     nI: 1 % number of input
                 nI_ref: 0 % **unused
                  nI_fb: 1 % number of feedbacks
                 method: 'exact-star'
          plantReachSet: {}
     controllerReachSet: {}
               numCores: 1
                  ref_I: []
               init_set: []
              reachTime: 0
              simTraces: {}
          controlTraces: {}
          falsifyTraces: {}
            falsifyTime: 0
```

IV.3.4 Constructing an NNV continuous nonlinear NNCS

IV.3.4.1 Constructing an FFNN controller object

This is a construction of an FFNN object. Please refer to section III.2 for details on how this object is constructed.

IV.3.4.2 Constructing a Continuous Nonlinear Plant Object

We use the *NonLinearODE* class to construct a continuous nonlinear plant object. A nonlinear continuous plant is defined as:

$$\dot{x}(t) = f(x,u,t), \; y(t) = Cx(t). \tag{IV.3}$$

where $x(t)$ is the state vector, $u(t)$ is the control input vector, $y(t)$ is the output vector, and C is the output matrix.

The *NonLinearODE* class takes the following parameters as inputs:

1. The *number of states*.
2. The *number of control inputs*.
3. The *dynamics function* $f(x,u,t)$.
4. The *reachability time step* of the plant.
5. The *control period* of the plant.
6. The *output matrix* defining the output vector of the plant.

In the following example, we construct a continuous nonlinear car model with 6 states and 1 control input. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_nncs_construct_nonlinearODE.m.

Code 36: Constructing a Continuous Nonlinear Plant

```
/* An example of constructing a continuous nonlinear plant */
Tr = 0.01; % reachability time step for the plant
Tc = 0.1; % control period of the plant
% output matrix
C = [0 0 0 0 1 0;1 0 0 -1 0 0; 0 1 0 0 -1 0]; % output matrix
car = NonLinearODE(6, 1, @car_dynamics, Tr, Tc, C);

function [dx]=car_dynamics(t,x,a_ego)
% note: t need to be here to do reachability
        mu=0.0001; % friction parameter

        % x1 = lead_car position
        % x2 = lead_car velocity
        % x3 = lead_car internal state
        % x4 = ego_car position
        % x5 = ego_car velocity
        % x6 = ego_car internal state

        % lead car dynamics
        a_lead = -5;
        dx(1,1)=x(2);
        dx(2,1) = x(3);
        dx(3,1) = -2 * x(3) + 2 * a_lead - mu*x(2)^2;
        % ego car dyanmics
        dx(4,1)= x(5);
        dx(5,1) = x(6);
        dx(6,1) = -2 * x(6) + 2 * a_ego - mu*x(5)^2;
end
```

Code 37: Results

```
car =

  NonLinearODE with properties:

                  options: [1x1 struct] % reach parameters
            dynamics_func: @car_dynamics
                      dim: 6 % number of states
                       nI: 1 % number of control inputs
                       nO: 3 % number of outputs
                        C: [3x6 double] % output matrix
    intermediate_reachSet: []
```

IV.3.4.3 Constructing a Continuous Nonlinear NNCS Object

After constructing an FFNN controller and a continuous nonlinear plant object, a continuous nonlinear NNCS object can be constructed by feeding the FFNN controller object and the plant object into the constructor of the *NonlinearNNCS* class.

The following example constructs an NNV continuous nonliner NNCS object. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_nncs_construct_nonlinearNNCS.m.

Code 38: Constructing a Continuous Nonlinear NNCS Object

```
/* An example of constructing a continuous nonlinear NNCS */
/ FFNN controller
load controller_5_20.mat;
weights = network.weights;
bias = network.bias;
n = length(weights);
Layers = [];
for i=1:n - 1
    L = LayerS(weights{1, i}, bias{i, 1}, 'poslin');
    Layers = [Layers L];
end
L = LayerS(weights{1, n}, bias{n, 1}, 'purelin');
Layers = [Layers L];
controller = FFNNS(Layers);
/* car model
Tr = 0.01; % reachability time step for the plant
Tc = 0.1; % control period of the plant
% output matrix
C = [0 0 0 0 1 0;1 0 0 -1 0 0; 0 1 0 0 -1 0]; % output matrix
car = NonLinearODE(6, 1, @car_dynamics, Tr, Tc, C);

ncs = NonlinearNNCS(controller, car); % system
```

Code 39: Results

```
ncs =

  NonlinearNNCS with properties:

              controller: [1x1 FFNNS]
                   plant: [1x1 NonLinearODE]
             feedbackMap: 0 % ** unused
                      nO: 3 % number of outputs
                      nI: 5 % number of inputs
                  nI_ref: 2 % number of reference inputs
                   nI_fb: 3 % number of feedback outputs
                   ref_I: [] % reference input to controller
                init_set: [] % initial set of states of the plant
            reachSetTree: []
       totalNumOfReachSet: 0
               reachTime: 0
              controlSet: []
                simTrace: []
            controlTrace: []
```

IV.3.5 Constructing a Discrete Nonlinear NNCS

IV.3.5.1 Constructing a FFNN controller object

This is a construction of an FFNN object. Please refer to section III.2 for details on how this object is constructed.

IV.3.5.2 Constructing a Discrete Nonlinear Plant Object

We use the *DNonLinearODE* class to construct a discrete nonlinear plant object. A discrete nonlinear plant is defined as:

$$x[k+1] = f(x[k], u[k]),\ y(k) = Cx(k). \tag{IV.4}$$

where $x[k]$ is the state vector, $u[k]$ is the control input vector, $y[k]$ is the output vector, and C is the output matrix.

The *DNonLinearODE* class takes the *number of states*, the *number of control inputs*, the *dynamics function* $f(x[k],u[k])$, and the *sampling period* T_s as inputs. The users also need to set the *output_mat* matrix C defining the outputs that are feedback to the controller. In the following example, we construct a discrete nonlinear mountain car model with 2 states and 1 control input. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_nncs_construct_dnonlinearODE.m.

Code 40: Constructing a Discrete Nonlinear Plant Object

```
/* An example of constructing a discrete nonlinear plant */
Ts = 0.5; % sampling time
C = [1 0; 0 1]; % output matrix
Car = DNonLinearODE(2, 1, @discrete_car_dynamics, Ts, C);

function [dx]=discrete_car_dynamics(t,x,u,T)
        % Note that t and T is required for reachability
        T = [];
        dx(1,1)=x(1) + x(2);
        dx(2,1)= -0.0025*cos(3*x(1)) + 0.0015 * u + x(2);
end
```

Code 41: Results

```
Car =

  DNonLinearODE with properties:

                options: [1x1 struct]

          dynamics_func: @discrete_car_dynamics

                    dim: 2 % number of states

                     nI: 1 % number of inputs

                     nO: 2 % number of outputs

                      C: [2x2 double] % output matrix

                     Ts: 0.5000 % sampling period

    intermediate_reachSet: []
```

IV.3.5.3 Constructing a Discrete Nonlinear NNCS Object

After constructing an FFNN controller and a discrete nonlinear plant object, a discrete nonlinear NNCS object can be constructed by feeding the FFNN controller object and the plant object into the constructor of the *DNonlinearNNCS* class.

The following example constructs an NNV discrete nonlinear NNCS object. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_nncs_construct_dnonlinearNNCS.m.

Code 42: Constructing a Discrete Nonlinear NNCS

```
/* An example of constructing a discrete nonlinear NNCS */
/* FFNN controller
load MountainCar_ReluController.mat;
W = nnetwork.W; % weight matrices
b = nnetwork.b; % bias vectors
n = length(W);
Layers = [];
for i=1:n - 1
    L = LayerS(W{1, i}, b{1, i}, 'poslin');
    Layers = [Layers L];
end
L = LayerS(W{1, n}, b{1, n}, 'purelin');
Layers = [Layers L];
controller = FFNNS(Layers);

/* MountainCar
Ts = 0.5; % sampling time
C = [1 0; 0 1]; % output matrix
Car = DNonLinearODE(2, 1, @discrete_car_dynamics, Ts, C);

ncs = DNonlinearNNCS(controller, Car); % system
```

Code 43: Result

```
ncs =
  DNonlinearNNCS with properties:
              controller: [1x1 FFNNS]
                   plant: [1x1 DNonLinearODE]
             feedbackMap: 0 % **unused
                      nO: 2 % number of outputs
                      nI: 2 % number of inputs
                  nI_ref: 0 % number of reference inputs
                   nI_fb: 2 % number of feedback outputs
                   ref_I: [] % reference input set
                init_set: [] % initial set of states of the plant
             reachSetTree: []
      totalNumOfReachSet: 0
               reachTime: 0
              controlSet: []
                simTrace: []
            controlTrace: []
```

IV.4 Specifying a property of an NNCS

After constructing an NNCS, the users need to specify the property of the system that they want to verify. The property is a linear predicate over the states/outputs of the system, i.e., the states/outputs of the plant model which is defined in the form of $P \triangleq Gy \leq g$, where y is the output vector of the system. Let P be an unsafe region, if the reachable sets of the system reach the unsafe region, the system is unsafe, otherwise, it is safe. In NNV, we use a *HalfSpace* object to represent a property. An example of constructing a property for the car in section IV.3.2 is given as follows.

Code 44: Specifying an NNCS property

```
/* An example of specifying an NNCS property */
t_gap = 1.4;
D_default = 10;
% safety specification:
%           x_lead - x_ego > t_gap * v_ego + D_default
% unsafe region: x_lead - x_ego - t_gap * v_ego <= D_default
unsafe_mat = [1 0 0 -1 -t_gap 0];
unsafe_vec = [D_default];
U = HalfSpace(unsafe_mat, unsafe_vec); % unsafe property
---------------------------------------------------------
Result
---------------------------------------------------------
U =
  HalfSpace with properties:
      G: [1 0 0 -1 -1.4000 0] % unsafe matrix
      g: 10 % unsafe vector
    dim: 6  % dimension
```

IV.5 Choosing a reachability method for an NNCS

For a continuous/discrete linear NNCS, NNV supports the *"exact-star"* and the *"approx-star"* reachability methods. The *"exact-star"* computes the exact reachable sets of the systems for a bounded time steps while the other computes an over-approximate reachable sets of the systems. For a continuous/discrete nonlinear NNCS, NNV supports the *"approx-star"* reachability method since we cannot compute the exact reachable set of a nonlinear plant model.

IV.6 Constructing an initial set of states for an NNCS

The initial set of states of the plant of an NNCS needs to be a *star set*. Instructions on how to construct a star set are given in section III.5.1.

IV.7 Choosing the number of cores utilized in computation

For an NNCS, NNV computes the reachable sets of the controller, then these reachable sets are fed to the plant as input sets. The reachable sets of the plant are then computed and fed back to the controller. To reduce conservativeness in the reachable set computation, NNV always computes the exact reachable sets for the controller (assumes it has piecewise-linear activation functions). Therefore, to speed up the computation, parallel computing is used by setting the *number of cores* we want to use for the computation.

IV.8 Verifying an NNCS

IV.8.1 Verifying a Continuous Linear NNCS

Users can verify a continuous linear NNCS using the *"verify"* method in the *LinearNNCS* class. The *verify* method takes *reachability parameters (reachPRM)* and a (unsafe) property as inputs. The *reachPRM* is a struct containing 5 parameters including:

1. *init_set* is the initial set of states of the plant.
2. *ref_input* is the reference input to the controller (no reference input: $ref\_input = [\,]$).
3. *numSteps* is the number of steps we want to verify.
4. *reachMethod* is the reachability method used for verification.
5. *numCores* is the number of cores used for computation.

The *verify* method returns:

1. *safe* is the safety result which can be *"SAFE"*, *"UNSAFE"* or *"UNKNOWN"*.
2. *counterExamples* which may be an array of star set counterexamples or falsified input points.
3. *verifyTime* is the verification time.

In the following example, we verify safety of a continuous, linear neural network addaptive cruise control sytem. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_nncs_verify_linearNNCS.m.

Code 45: Verifying A Continuous Linear NNCS

```
/* An example of verifying a continuous linear NNCS */
/* Controller
load controller_5_20.mat; weights = network.weights;
bias = network.bias; n = length(weights); Layers = [];
for i=1:n - 1
    L = LayerS(weights{1, i}, bias{i, 1}, 'poslin');
    Layers = [Layers L];
end
L = LayerS(weights{1, n}, bias{n, 1}, 'purelin');
Layers = [Layers L];
Controller = FFNNS(Layers);

/* plant model
A = [0 1 0 0 0 0 0; 0 0 1 0 0 0 0; 0 0 0 0 0 0 1; ...
    0 0 0 0 1 0 0; 0 0 0 0 0 1 0; 0 0 0 0 0 -2 0; ...
    0 0 -2 0 0 0 0];
B = [0; 0; 0; 0; 0; 2; 0];
C = [1 0 0 -1 0 0 0; 0 1 0 0 -1 0 0; 0 0 0 0 1 0 0];
D = [0; 0; 0];
Tc = 0.1; % control period
Nr = 20; % number of reachability steps in 1 control period
plant = LinearODE(A, B, C, D, Tc, Nr); % continuous plant model

/* continuous linear NNCS
ncs = LinearNNCS(Controller, plant); % a continuous linear NNCS
...
```

Code 46: Verifying a Continuous Linear NNCS(cont.)

```
...
/* ranges of initial set of states of the plant
lb = [90; 29; 0; 30; 30; 0; -10];
ub = [92; 30; 0; 31; 30.5; 0; -10];

/* reachability parameters
reachPRM.init_set = Star(lb, ub);
reachPRM.ref_input = [30; 1.4];
reachPRM.numSteps = 10;
reachPRM.reachMethod = 'approx-star';
reachPRM.numCores = 4;

/* usafe region: x1 - x4 <= 1.4 * v_ego + 10
unsafe_mat = [1 0 0 -1 -1.4 0 0];
unsafe_vec = 10;
U = HalfSpace(unsafe_mat, unsafe_vec);

/* verify the system
[safe, counterExamples, verifyTime] = ncs.verify(reachPRM, U);
---------------------------------------------------------------
Results
---------------------------------------------------------------
safe = 'SAFE';
counterExamples = [];
verifyTime = 3.6339;
```

IV.8.2 Verifying a Discrete Linear NNCS

Users can verify a discrete linear NNCS using the *"verify"* method in the *DLinearNNCS* class. The *verify* method takes *reachability parameters (reachPRM)* and an (unsafe) property as inputs. The *reachPRM* is a struct containing 5 parameters including:

1. *init_set* is the initial set of states of the plant.
2. *ref_input* is the reference input to the controller (no reference input: $ref\_input = [\,]$).
3. *numSteps* is the number of steps we want to verify.
4. *reachMethod* is the reachability method used for verification.
5. *numCores* is the number of cores used for computation.

The *verify* method returns:

1. *safe* is the safety result which can be *"SAFE"*, *"UNSAFE"* or *"UNKNOWN"*.
2. *counterExamples* which may be an array of star set counterexamples or falsified input points.
3. *verifyTime* is the verification time.

In the following example, we verify safety of a discrete, linear neural network addaptive cruise control sytem. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_nncs_verify_dlinearNNCS.m.

Code 47: Verifying a Discrete Linear NNCS

```
/* An example of verifying a discrete linear NNCS */
/* Controller
load controller_5_20.mat; weights = network.weights;
bias = network.bias; n = length(weights); Layers = [];
for i=1:n - 1
    L = LayerS(weights{1, i}, bias{i, 1}, 'poslin');
    Layers = [Layers L];
end
L = LayerS(weights{1, n}, bias{n, 1}, 'purelin');
Layers = [Layers L];
Controller = FFNNS(Layers);

/* plant model
A = [0 1 0 0 0 0 0; 0 0 1 0 0 0 0; 0 0 0 0 0 0 1; ...
    0 0 0 0 1 0 0; 0 0 0 0 0 1 0; 0 0 0 0 0 -2 0; ...
    0 0 -2 0 0 0 0];
B = [0; 0; 0; 0; 0; 2; 0];
C = [1 0 0 -1 0 0 0; 0 1 0 0 -1 0 0; 0 0 0 0 1 0 0];
D = [0; 0; 0];
plant = LinearODE(A, B, C, D); % continuous plant model
plantd = plant.c2d(0.1); % discrete plant model

/* discrete linear NNCS
ncs = DLinearNNCS(Controller, plantd); % a discrete linear NNCS
...
```

Code 48: Verifying a Discrete Linear NNCS (cont.)

```
...
/* ranges of initial set of states of the plant
lb = [90; 29; 0; 30; 30; 0; -10];
ub = [92; 30; 0; 31; 30.5; 0; -10];

/* reachability parameters
reachPRM.init_set = Star(lb, ub);
reachPRM.ref_input = [30; 1.4];
reachPRM.numSteps = 10;
reachPRM.reachMethod = 'approx-star';
reachPRM.numCores = 4;

/* usafe region: x1 - x4 <= 1.4 * v_ego + 10
unsafe_mat = [1 0 0 -1 -1.4 0 0];
unsafe_vec = 10;
U = HalfSpace(unsafe_mat, unsafe_vec);

/* verify the system
[safe, counterExamples, verifyTime] = ncs.verify(reachPRM, U);
---------------------------------------------------------------
Results
---------------------------------------------------------------
safe = 'SAFE';
counterExamples = [];
verifyTime = 1.8996;
```

IV.8.3 Verifying a Continuous Nonlinear NNCS

Users can verify a continuous nonlinear NNCS using the *"verify"* method in the *NonLinearNNCS* class. The *verify* method takes *reachability parameters (reachPRM)* and a (unsafe) property as inputs. The *reachPRM* is a struct containing 5 parameters including:

1. *init_set* is the initial set of states of the plant.
2. *ref_input* is the reference input to the controller (no reference input: $ref\_input = [\,]$).
3. *numSteps* is the number of steps we want to verify.
4. *reachMethod* is the reachability method used for verification. Always need to be *"approx-star"*.
5. *numCores* is the number of cores used for computation.

The *verify* method returns:

1. *safe* is the safety result which can be *"SAFE"*, *"UNSAFE"* or *"UNKNOWN"*.
2. *counterExamples* which may be an array of star set counterexamples or falsified input points.
3. *verifyTime* is the verification time.

In the following example, we verify safety of a continuous, nonlinear neural network adaptive cruise control sytem. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_nncs_verify_nonlinearNNCS.m.

Code 49: Verifying a Continuous Nonlinear NNCS

```
/* An example of verifying a continuous nonlinear NNCS */
/* FFNN controller
load controller_5_20.mat;
weights = network.weights;
bias = network.bias;
n = length(weights);
Layers = [];
for i=1:n - 1
    L = LayerS(weights{1, i}, bias{i, 1}, 'poslin');
    Layers = [Layers L];
end
L = LayerS(weights{1, n}, bias{n, 1}, 'purelin');
Layers = [Layers L];
controller = FFNNS(Layers);

/* car model
Tr = 0.01; % reachability time step for the plant
Tc = 0.1; % control period of the plant
% output matrix
C = [0 0 0 0 1 0;1 0 0 -1 0 0; 0 1 0 0 -1 0]; % output matrix
car = NonLinearODE(6, 1, @car_dynamics, Tr, Tc, C);

/* system
ncs = NonlinearNNCS(controller, car);
...
```

Code 50: Verifying a Continuous Nonlinear NNCS (cont.)

```
...
/* ranges of initial set of states of the plant
lb = [90; 29; 0; 30; 30; 0];
ub = [92; 30; 0; 31; 30.5; 0];

/* reachability parameters
reachPRM.init_set = Star(lb, ub);
reachPRM.ref_input = [30; 1.4];
reachPRM.numSteps = 50;
reachPRM.reachMethod = 'approx-star';
reachPRM.numCores = 4;
/* usafe region: x1 - x4 <= 1.4 * v_ego + 10
unsafe_mat = [1 0 0 -1 -1.4 0];
unsafe_vec = 10;
U = HalfSpace(unsafe_mat, unsafe_vec);

/* verify the system
[safe, counterExamples, verifyTime] = ncs.verify(reachPRM, U);
-------------------------------------------------------------
Results
-------------------------------------------------------------
safe = "UNSAFE";
counterExamples = 1000 counterExamples are found
verifyTime = 88.96
```

IV.8.4 Verifying a Discrete Nonlinear NNCS

Users can verify a discrete nonlinear NNCS using the *"verify"* method in the *DNonLinearNNCS* class. The *verify* method takes *reachability parameters (reachPRM)* and a (unsafe) property as inputs. The *reachPRM* is a struct containing 5 parameters including:

1. *init_set* is the initial set of states of the plant.
2. *ref_input* is the reference input to the controller (no reference input: $ref\_input = [\,]$).
3. *numSteps* is the number of steps we want to verify.
4. *reachMethod* is the reachability method used for verification. Always need to be *"approx-star"*.
5. *numCores* is the number of cores used for computation.

The *verify* method returns:

1. *safe* is the safety result which can be *"SAFE"*, *"UNSAFE"* or *"UNKNOWN"*.
2. *counterExamples* which may be an array of star set counterexamples or falsified input points.
3. *verifyTime* is the verification time.

In the following example, we verify safety of a discrete, nonlinear neural network mountain car sytem. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_nncs_verify_dnonlinearNNCS.m.

Code 51: Verifying a Discrete Nonlinear NNCS

```
/* An example of constructing a discrete nonlinear NNCS */
/* FFNN controller
load MountainCar_ReluController.mat;
W = nnetwork.W; % weight matrices
b = nnetwork.b; % bias vectors
n = length(W);
Layers = [];
for i=1:n - 1
    L = LayerS(W{1, i}, b{1, i}, 'poslin');
    Layers = [Layers L];
end
L = LayerS(W{1, n}, b{1, n}, 'purelin');
Layers = [Layers L];
controller = FFNNS(Layers);

/* MountainCar
Ts = 0.5; % sampling time
C = [1 0; 0 1]; % output matrix
Car = DNonLinearODE(2, 1, @discrete_car_dynamics, Ts, C);

ncs = DNonlinearNNCS(controller, Car); % system
...
```

Code 52: Verifying a Discrete Nonlinear NNCS (cont.)

```
...
b = [-0.41; 0];
ub = [0.4; 0];
reachPRM.init_set = Star(lb, ub);
reachPRM.ref_input = [];
reachPRM.numSteps = 10;
reachPRM.reachMethod = 'approx-star';
reachPRM.numCores = 4;

% unsafe region
U = HalfSpace([-1 0], 0); % x1 > 0

[safe, counterExamples, verifyTime] = ncs.verify(reachPRM, U);
--------------------------------------------------------------
Results
--------------------------------------------------------------
safe = "UNSAFE";
counterExamples = 485 counterExamples are found
verifyTime = 4.2645
```

IV.9 Visualizing the results

For a linear NNCS, user can visualize the reachable sets of the sytem using the *"plotOutputReachSets"* method in the *LinearNNCS* or *DLinearNNCS* class. This method plots the reachable set of the system in a specific direction defined by the *mapping matrix* G and the *offset vector* g. Mathematically, if the state vector of the system is x, the method plots $y = G \times x + g$. The users can also specify the color of the reachable sets they want to plot.

In the following example, we plot the reachable sets of the continuous linear neural network addaptive cruise control systems in the example of section IV.8.1. The code for this example is available at https://github.com/verivital/nnv/code/example/Manual/example_nncs_visualize_linearNNCS.m.

Code 53: Visualizing Reachable Sets of Linear NNCS

```
...
/* verify the system
[safe, counterExamples, verifyTime] = ncs.verify(reachPRM, U);
/* Plot output reach sets: actual distance vs. safe distance
% plot reachable set of the distance between two cars
figure;
map_mat = [1 0 0 -1 0 0 0];
map_vec = [];
ncs.plotOutputReachSets('blue', map_mat, map_vec);
hold on;
% plot safe distance between two cars:
% d_safe = D_default + t_gap * v_ego;
% D_default = 10; t_gap = 1.4, d_safe = 10 + 1.4 * x5;
map_mat = [0 0 0 0 1.4 0 0];
map_vec = [10];
ncs.plotOutputReachSets('red', map_mat, map_vec);
title('Actual Distance versus. Safe Distance');

/* plot 2d output sets
figure;
map_mat = [1 0 0 -1 0 0 0; 0 0 0 0 1 0 0]; map_vec = [];
ncs.plotOutputReachSets('blue', map_mat, map_vec);
title('Actual Distance versus. Ego car speed');
```

Figures IV.2 and IV.3 illustrate the reachable sets of the system. One can observe that the actual distance $>$ the safe distance, thus, the system is safe (in 10 control periods).

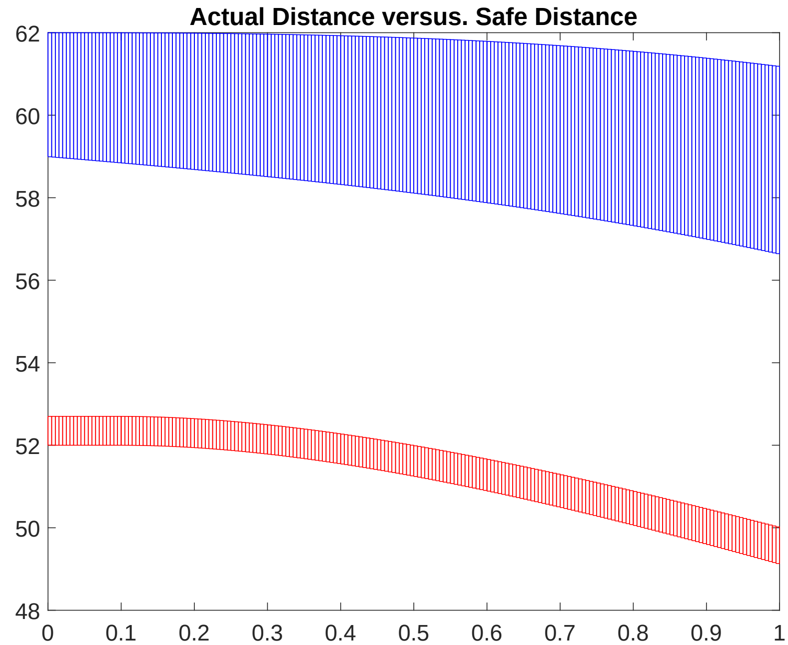

Figure IV.2: Reachable set of actual distance vs. the safe distance over time.

Similarly for the discrete linear neural network adaptive cruise control system verified in section IV.8.2, we can plot Figures IV.4 and IV.5 using the script that is available at https://github.com/verivital/nnv/code/example/Manual/example_nncs_visualize_dlinearNNCS.m.

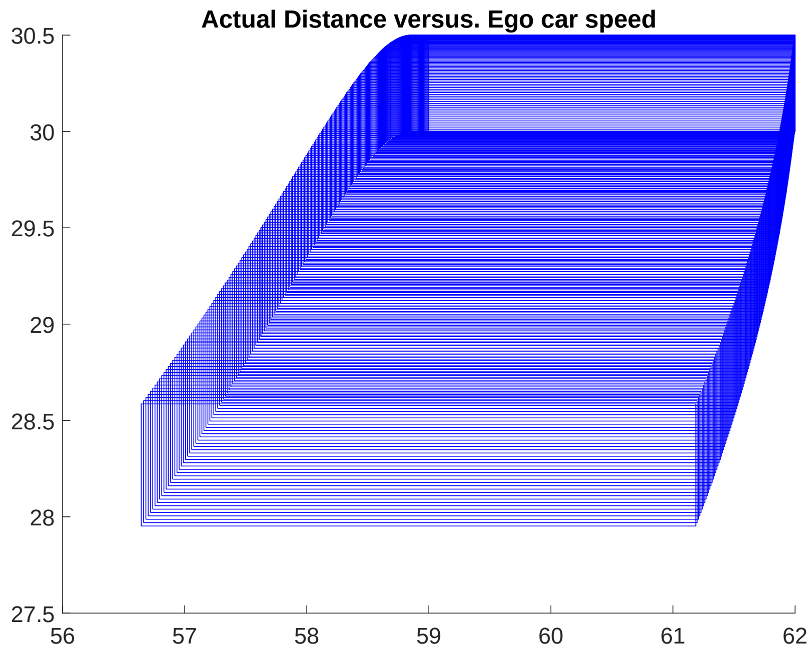

Figure IV.3: Reachable set of actual distance and the velocity of the ego car.

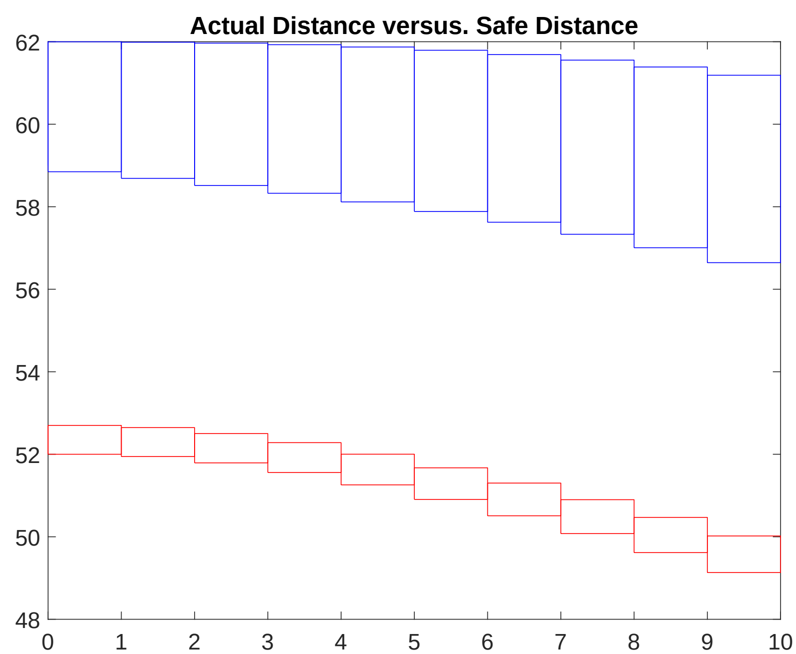

Figure IV.4: Reachable set of actual distance vs. the safe distance over time.

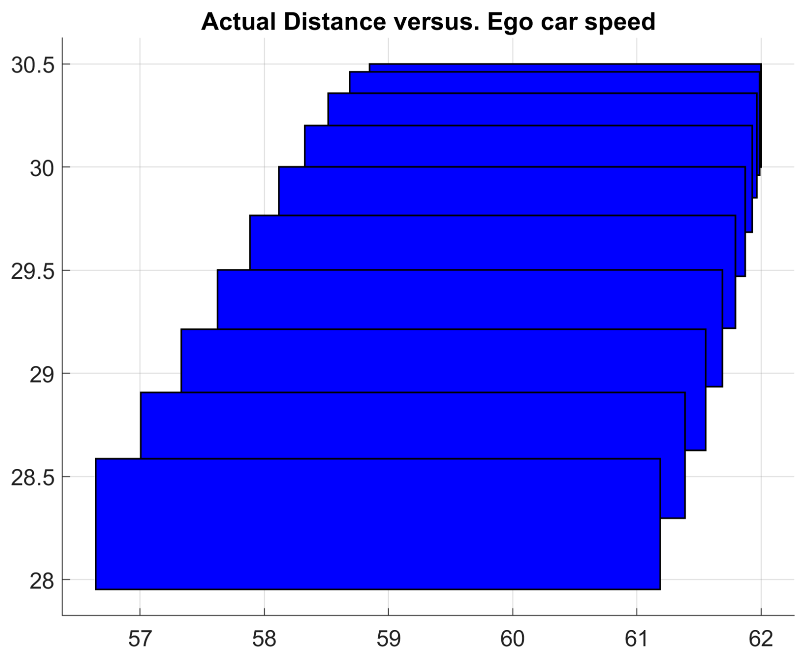

Figure IV.5: Reachable set of actual distance and the velocity of the ego car.

BIBLIOGRAPHY

Althoff, M. (2015). An introduction to cora 2015. In *Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems*.

Bak, S., Bogomolov, S., and Johnson, T. T. (2015). Hyst: a source transformation and translation tool for hybrid automaton models. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 128–133. ACM.

Kvasnica, M., Grieder, P., Baotić, M., and Morari, M. (2004). Multi-parametric toolbox (mpt). In *International Workshop on Hybrid Systems: Computation and Control*, pages 448–462. Springer.