StarV

User Manual

By: Yuntao Li, Sung Woo Choi, Qing Liu, Dung Hoang Tran

Contents

List of Figures

List of Tables

Chapter 1

Overview of StarV and its Features

1 Overview

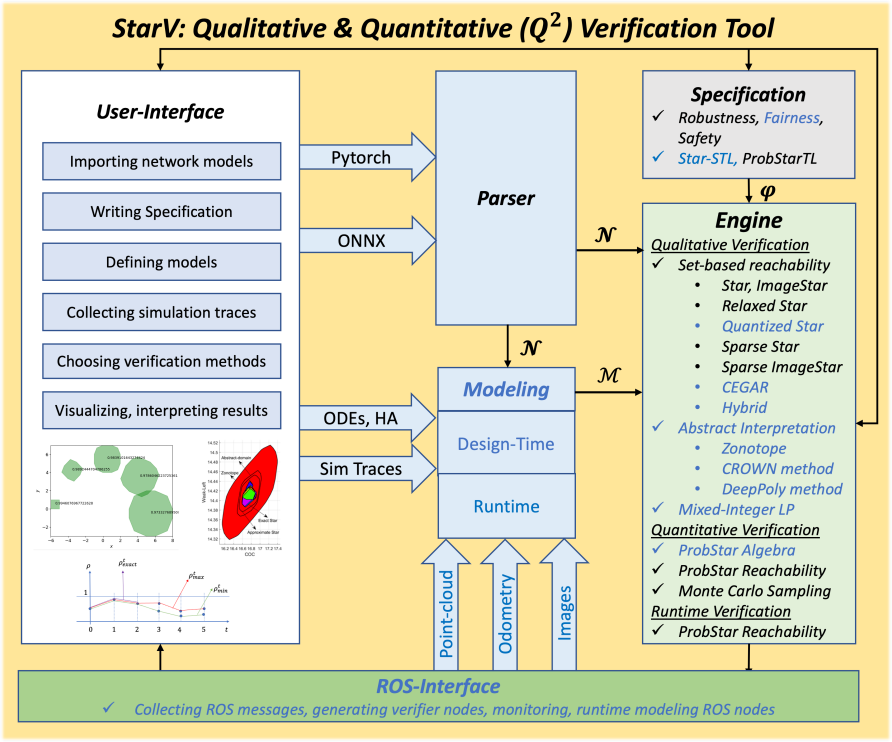


Figure 1.1: A conceptual overview of StarV and core features (Features in blue color are under development).

StarV is an object-oriented Python toolbox that aims to integrate scalable verification techniques for LES, including reachability-based [11], abstract interpretation [7,12], and mixed integer linear programming [4]. It is designed not only for academic research but also to facilitate the adoption of formal methods in real-world robotic applications built on ROS [2,6]. Figure 1.1 depicts its conceptual overview and significant features.

Conceptually, StarV is organized into six modules: a user interface, a parser, a specification module, a modeling module, and an engine module, with a ROS interface currently under development. The user interface enables users to import network models, write specifications, define models, collect simulation traces, choose verification methods, and visualize results. The parser automatically converts Pytorch and ONNX [5] models into StarV's internal representations, though users can also define custom models using supported layers and plant objects. The specification module supports the formulation of safety, robustness, and temporal properties (via ProbStar Temporal Logic, ProbStarTL), while fairness and Star temporal logic (StarTL) for deterministic input sets remain under development. The modeling module in StarV enables users to construct neural network control systems (NNCS) at both design and runtime. At design time, users can construct an NNCS with a neural network controller and a physical plant model (ODEs or Hybrid Automaton). We are also working to extend this capability to model complex learning-enabled cyber-physical systems (CPS) that incorporate multiple interacting neural network components. At runtime, we plan to support perception-based modeling where linear plant motion dynamics, for instance, human or vehicle motion [2,6], are derived solely from

ROS perception data such as LiDAR and camera point-clouds. Although these works have been done, they have not yet been fully integrated into StarV. In parallel, the ROS interface will be developed in the future to support the automatic generation of verifiers, runtime modeling, and monitoring ROS nodes for robotic applications. At the core of StarV, the engine module implements various set representations alongside multiple verification and reachability algorithms.

The verification workflow begins by constructing an LES model object, either a generic neural network or an NNCS. Next, users specify the desired property (safety, robustness, or temporal) and define the input conditions using sets such as Star, ImageStar, or ProbStar, along with verification parameters (e.g., number of cores, LP solver, or time steps for NNCS). Verification is then executed via methods provided by the StarV LES object, and the resulting reachable sets and outcomes are visualized with built-in plotting functions. Detailed workflows for various LES configurations can be found in the later chapters of the user manual.

2 Features

Table 1 summarizes StarV's core features, highlighting its novel contributions. StarV introduces the probabilistic star (ProbStar) representation for efficient reachability analysis, enabling both exact and approximate quantitative verification of safety properties. It further extends verification capabilities with ProbStar Temporal Logic (ProbStarTL), which transforms temporal specifications into a computable disjunctive normal form for precise or bounded probability estimation. For massive linear systems, StarV leverages simulation-based methods using the Krylov subspace technique and state-space projection to enhance memory and computation efficiency. Additionally, the toolbox presents SparseImageStar, a memory-efficient approach for verifying deep CNNs that preserves spatial relationships through an indices-shifting technique, thereby facilitating novel sparse convolution and pooling operations.

Feature	**Supported**
Neural Network Type	FFNN, CNN, SSN, Vanilla RNN, LSTM, GRU
Layers	MaxPool, Conv, BN, AvgPool, FC, TC, DC,
Activation functions	ReLU, Satlin, Sigmoid, Tanh, Leaky ReLU, Satlins
Plant dynamics (NNCS)	Linear ODE, Massive Linear ODE, Continuous & Discrete Time
Set Representation	Star, ImageStar, SparseStar, SparseImageStar, ProbStar
Qualitative Reach methods	exact, approx, relax, abs-dom
Quantitative Reach methods	exact, approx
Reachable set visualization	exact and over-approximation
Specification	Safety, Robustness, ProbStarTL Temporal Properties
Miscellaneous	Parallel computing, counterexample generation
Solver	Gurobi, GLPK
Import	Pytorch, ONNX

Table 1.1: Overview of core features available in StarV. BN refers to batch normalization layers, FC to fully-connected layers, AvgPool to average pooling layers, Conv to convolutional layers, MaxPool to max-pooling layers, TC to transpose convolutional layers, and DC to dilated convolutional layers.

Chapter 2

Installation

1 Overview

The StarV tool comprises the following directories:

`StarV/`	Root directory
`StarV/README.md`	Project overview
`StarV/requirements.txt`	List of dependencies
`StarV/gurobi.lic`	License file for Docker usage
`StarV/setup.py`	Setup script for the Python package
`StarV/.devcontainer/`	Docker scripts for development
`StarV/StarV/`	Main algorithm scripts
`StarV/artifacts/`	Artifacts directory
`StarV/tests/`	Testing scripts
`StarV/tutorials/`	Tutorials and examples

2 Installation and Setup

StarV has been tested on Ubuntu 20.04, 22.04, and Python 3.8+. Other versions may work but are untested. Before you start, please remove any previous copy of StarV from your system before installing a new version!

2.1 Cloning the Repository

We provide two options to install StarV tool. To start, clone the StarV repository:

```
git clone https://github.com/V2A2/StarV
```

2.2 Option 1: Running Locally

Installing Gurobi on Ubuntu

1. Download and Extract Gurobi. Visit https://www.gurobi.com/downloads/ to download Gurobi or use the command below:

```
wget https://packages.gurobi.com/10.0/gurobi10.0.1_linux64.tar.gz
```

Move the downloaded file to `/opt` (you may need to create this directory):

```
mv gurobi10.0.1_linux64.tar.gz ~/opt/
cd ~/opt/
tar -xzvf gurobi10.0.1_linux64.tar.gz
rm gurobi10.0.1_linux64.tar.gz
```

2. Set Environment Variables. Edit the ~/.bashrc file to include the following, replacing {PATH_TO_YOUR_HOME} with the absolute path to your home directory:

```
export GUROBI_HOME="{PATH_TO_YOUR_HOME}/opt/gurobi1001/linux64"
export GRB_LICENSE_FILE="{PATH_TO_YOUR_HOME}/gurobi.lic"
export PATH="${PATH}:${GUROBI_HOME}/bin"
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:${GUROBI_HOME}/lib"
```

Reload the configuration:

```
source ~/.bashrc
```

3. Obtain a License. Visit https://www.gurobi.com/academia/academic-program-and-licenses/ to acquire your license. Execute the grbgetkey command provided on the website to activate your license. At ~/opt/gurobi1001/linux64/bin, copy the grbgetkey command from the site and enter it into a terminal. Save the Gurobi license gurobi.lic in the corresponding directory to GRB_LICENSE_FILE.

Installing StarV and Dependencies

1. Install System Packages. Run the following command to install Ubuntu packages and dependencies:

```
sudo apt-get install python3-dev python3-pip libgmp-dev libglpk-dev libgmp3-dev
```

2. Set up Conda Environment. StarV is tested on Python 3.8+. It can be installed easily into a conda environment. If you don't have conda, you can install miniconda.

```
conda deactivate; conda env remove -n starv  # Remove old environment if necessary
conda create -n starv python=3.8             # Create new environment
conda activate starv                         # Activate environment
```

3. Install StarV. Install StarV as a local Python package. Dependencies in requirements.txt will be installed automatically. Run the following command at the /StarV root directory:

```
pip3 install -e .
pip3 show starv # Check for installation
```

4. Uninstall StarV. StarV can be easily uninstalled within the conda environment by using the following command:

```
pip3 uninstall starv
```

Python Dependencies in requirements.txt:

gurobipy==11.0.3, glpk, pycddlib<=2.1.8, polytope, pypoman, tabulate, matplotlib, numpy<=1.26.4, scipy<1.13.1, ipyparallel, torchvision plotly==5.14.1, onnx, onnx2pytorch, onnxruntime, scikit-learn

2.3 Option 2: Running with Docker

1. Acquire a Gurobi Web License Service (WLS) License Visit https://www.gurobi.com/features/web-license-service/ and save the gurobi.lic file at the /StarV root directory:

```
StarV/
|   gurobi.lic
|   ...
|---.devcontainer/
|   |   Dockerfile
|   |   build_docker.sh
|   |   launch_docker.sh
```

2. Build and Launch Docker Run the following commands to build and launch the Docker container:

```
sh .devcontainer/build_docker.sh
sh .devcontainer/launch_docker.sh
```

3 Testing the Installation

To verify the installation, execute any testing script from `StarV/tests/`. For example:

```
conda activate starv    # Activate conda if running locally
python3 tests/test_set_probstar.py
```

Chapter 3

Reachable Sets

1 Star

1.1 Star Set Construction

In StarV, we support multiple ways of constructing a star set (Def. 1). Some examples are provided below.

Definition 1 (Generalized Star Set [1,3,10])**.** A generalized star set *(or simply star)* Θ *is a tuple* $\langle c, V, P, l, u\rangle$ *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,* $P : \mathbb{R}^m \to \{\top, \bot\}$ *is a predicate,* l *and* u *are the lower-bound and upper-bound vectors of the predicate variables. The basis vectors are arranged to form the star's* $n \times m$ *basis matrix. The set of states represented by the star is given as:*

$$[\![\Theta]\!] = \{x \mid x = c + \Sigma_{i=1}^{m}(\alpha_i v_i),\ P(\alpha_1, \cdots, \alpha_m) = \top, l[i] \le \alpha_i \le u[i]\}.$$

For the sake of brevity, we refer to both the tuple Θ *and the set of states* $[\![\Theta]\!]$ *as* Θ*. In this work, we restrict the predicates to be a conjunction of linear constraints,* $P(\alpha) \triangleq C\alpha \le d$ *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.*

Proposition 1. *Any bounded convex polyhedron* $\mathcal{P} \triangleq \{x \mid Cx \le d, x \in \mathbb{R}^n\}$ *can be represented as a star.*

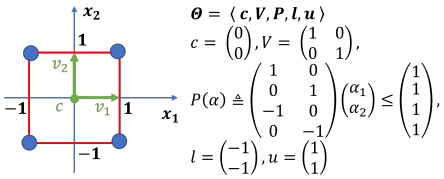


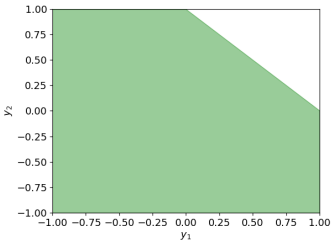
Figure 3.1: An example of a star set.

Method 1: Construction using basis matrix and constraints

In this method, the star set is constructed by explicitly specifying the basis matrix, predicate constraints, and predicate bounds. A 2D example is provided below:

```python
# Define center and basis vectors
c1 = np.array([[0], [0]])
v1 = np.array([[1], [0]])
v2 = np.array([[0], [1]])
V = np.hstack((c1, v1, v2))

# Define bounds and constraints
# Predicate bounds:
# -1 <= alpha_1 <= 1
# -1 <= alpha_2 <= 1
pred_lb = np.array([-1, -1])
pred_ub = np.array([1, 1])
# Constraints:
```

```
# repr(S)
Star Set:
V: (2, 3)
Predicate Constraints:
C: (1, 2)
d: (1,)
dim: 2
nVars: 2
pred_lb: (2,)
pred_ub: (2,)

# print(S)
Star Set:
V: [[0 1 0]
 [0 0 1]]
Predicate Constraints:
C: [[1 1]]
d: [1]
```



`plot_star(S)`

```
# alpha_1 + alpha_2 <= 1
C = np.array([[1, 1]])
d = np.array([1])

S = Star(V, C, d, pred_lb, pred_ub)
repr(S)
print(S)
plot_star(S)
```

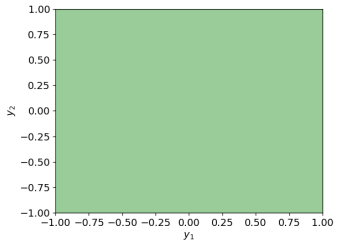
```
dim: 2
nVars: 2
pred_lb: [-1 -1]
pred_ub: [1 1]
```

Method 2: Construction using state bounds

This method creates a box-shaped star set by specifying state bounds directly:

```
# State bounds:
# -1 <= x1 <= 1
# -1 <= x2 <= 1
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
S = Star(lb, ub)

print(S)
plot_star(S)
```

```
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
```



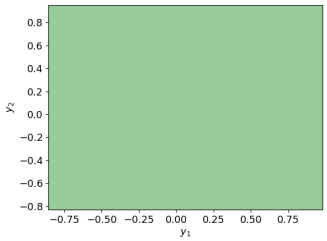
plot_star(S)

Method 3: Star set construction with random state bounds

This method generates a random 2D star set using state bounds:

```
# Generate a random 2D star
S = Star.rand(2)

print(S)
plot_star(S)
```

```
Star Set:
V: [[0.06217 0.91793 0.     ]
 [0.06076 0.      0.8913 ]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
```



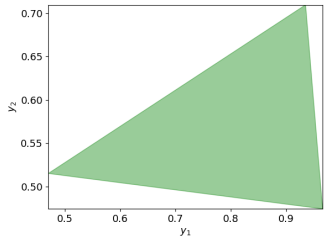
plot_star(S)

Method 4: Star set construction with random H-representation polytope

Here, a random 2D star set is generated with extra constraints from the H-representation polytope (e.g., 3 constraints):

```
# 2D random star with 3 constraints
S = Star.rand_polytope(2, 3)

print(S)
plot_star(S)
```

```
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[-0.082   -0.99663]
 [-0.38582  0.92257]
 [ 0.99158  0.1295 ]]
d: [-0.55185  0.29381  1.0196 ]
dim: 2
nVars: 2
pred_lb: [0.47005 0.47421]
pred_ub: [0.96632 0.70972]
```



plot_star(S)

1.2 Affine mapping

Proposition 2. *[Affine Mapping of a Star] Given a star set $\Theta = \langle c, V, P, l, u\rangle$, an affine mapping of the star Θ with the affine mapping matrix W and offset vector b defined by $\bar{\Theta} = \{y \mid y = Wx + b,\ x \in \Theta\}$ is another star with the following characteristics: $\bar{\Theta} = \langle \bar{c}, \bar{V}, \bar{P}, \bar{l}, \bar{u}\rangle$, $\bar{c} = Wc + b$, $\bar{V} = \{Wv_1, Wv_2, \cdots, Wv_m\}$, $\bar{P} \equiv P, \bar{l} \equiv l, \bar{u} \equiv u$.*

Below is a simplified code example demonstrating the affine mapping operation. In this example, a box-shaped star set is created and then transformed using an affine mapping defined by a transformation matrix W and an offset vector b defined by the user.

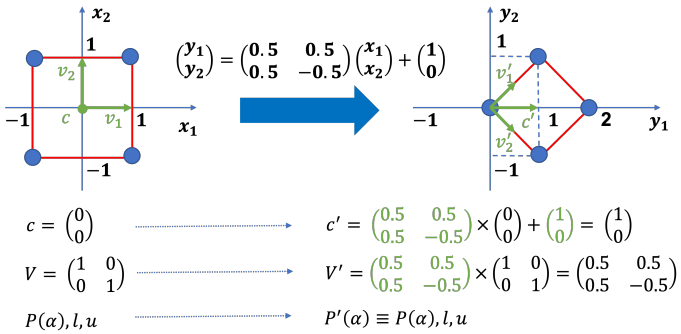
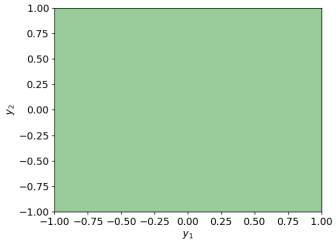


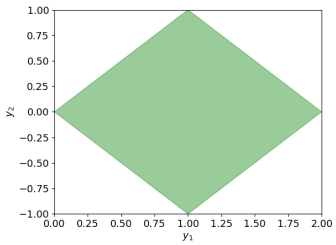
Figure 3.2: Affine mapping of a star set.

```python
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
S = Star(lb, ub)

print('\nOriginal star set:')
print(S)
plot_star(S)

# Define affine transformation:
# y = Wx + b
# Transformation matrix (scaling + rotation)
W = np.array([[0.5, 0.5],
              [0.5, -0.5]])
# Offset vector (translation)
b = np.array([1, 0])

# Apply affine mapping to obtain a new star set
S_mapped = S.affineMap(W, b)

print('\nAffine mapped star set:')
print(S_mapped)
plot_star(S_mapped)
```

```
Original star set:
# print(S)
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]

Affine mapped star set:
# print(S_mapped)
Star Set:
V: [[ 1.  2. -1.]
 [ 0.  1.  1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
```



plot_star(S)



plot_star(S_mapped)

1.3 Estimate ranges

Proposition 3. *[Estimated range of a state] Given a star set $\Theta = \langle c, V, P, l, u \rangle$, the range of the state vector x of the star set can be estimated quickly without solving the linear programming optimization problems by using only the lower bound and upper bound vectors of the predicate variables.*

$$x[i]^{est}_{min} = c[i] + \sum_{\substack{j=1 \\ v_j[i] \geq 0}}^{m} v_j[i] l[i] + \sum_{\substack{k=1 \\ v_k[i] \leq 0}}^{m} v_k[i] u[i],$$

$$x[i]^{est}_{max} = c[i] + \sum_{\substack{j=1 \\ v_j[i] \geq 0}}^{m} v_j[i] u[i] + \sum_{\substack{k=1 \\ v_k[i] \leq 0}}^{m} v_k[i] l[i].$$

In another way,

$$l_{est} \leq x = c + V\alpha = c + max(0, V)\alpha + min(0, V)\alpha \leq u_{est},$$
$$l_{est} = c + max(0, V)l + min(0, V)u,$$
$$u_{est} = c + max(0, V)u + min(0, V)l.$$

The following simplified code demonstrates how to estimate the ranges for a Star set. In this example, a 2D star set is created using given state bounds. The ranges of the state vector are then estimated both for a single dimension and for all dimensions using the methods `estimateRange(dim)` and `estimateRanges()` respectively.

```python
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
S = Star(lb, ub)

# 1. Estimate range for a single dimension (e.g., dimension 0)
dim = 0
est_min, est_max = S.estimateRange(dim)
print(f'\nEstimated range for dimension {dim}:')
print(f'Estimated lower state bound (l_est[{dim}]) = {est_min}')
print(f'Estimated upper state bound (u_est[{dim}]) = {est_max}')

# 2. Estimate ranges for all dimensions
est_mins, est_maxs = S.estimateRanges()
print('\nEstimated ranges for all dimensions:')
print(f'Estimated lower state bounds (l_est) = {est_mins}')
print(f'Estimated lower state bounds (u_est) = {est_maxs}')
```

```
Actual state bounds:
Lower bounds (l): [-1. -1.]
Upper bounds (u): [1. 1.]

Estimated range for dimension 0:
Estimated 0th lower state bound (l_est[0]) = -1.
Estimated 0th upper state bound (u_est[0]) = 1.

Estimated ranges for all dimensions:
Estimated lower state bounds (l_est) = [-1. -1.]
Estimated upper state bounds (u_est) = [1. 1.]
```

1.4 Optimized Ranges

Proposition 4. *[Optimized range of a state] Given a star set $\Theta = \langle c, V, P, l, u\rangle$, the range of the i^{th} state $x[i]$ of the star set can be found by solving the following linear programming optimization problems:*

$$x[i]_{min} = min(c[i] + \Sigma_{j=1}^{m} v_j[i]\alpha_j),\ s.t.\ P(\alpha) \triangleq C\alpha \leq d \wedge l \leq \alpha \leq u,$$
$$x[i]_{max} = max(c[i] + \Sigma_{j=1}^{m} v_j[i]\alpha_j),\ s.t.\ P(\alpha) \triangleq C\alpha \leq d \wedge l \leq \alpha \leq u.$$

The code below demonstrates how to compute the optimized ranges for a star set using LP solvers. StarV supports multiple LP solvers, such as gurobi by default, linprog, and glpk.

```python
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
S = Star(lb, ub)

# 1. Compute optimized range for a single dimension (e.g.,
↪  dimension 0)
dim = 0
opt_min = S.getMin(dim, 'gurobi')
opt_max = S.getMax(dim, 'gurobi')
print(f'\nOptimized range for dimension {dim}:')
print(f'Optimized lower state bound (x[{dim}]_min) =
↪  {opt_min:.6f}')
print(f'Optimized upper state bound (x[{dim}]_max) =
↪  {opt_max:.6f}')

# 2. Compute optimized ranges for all dimensions
opt_mins, opt_maxs = S.getRanges('gurobi')
print('\nOptimized ranges for all dimensions:')
print(f'Optimized lower state bounds (x_min) = {opt_mins}')
print(f'Optimized upper state bounds (x_max) = {opt_maxs}')
```

```
Actual state bounds:
Lower bounds (l): [-1. -1.]
Upper bounds (u): [1. 1.]

Optimized range for dimension 0:
Optimized lower state bound (x[0]_min) = -1.000000
Optimized upper state bound (x[0]_max) = 1.000000

Optimized ranges for all dimensions:
Optimized lower state bounds (x_min) = [-1. -1.]
Optimized upper state bounds (x_max) = [1. 1.]
```

1.5 Constraint operations

Proposition 5 (Star and Half-space Intersection)**.** *The intersection of a star $\Theta \triangleq \langle c, V, P, l, u\rangle$ and a half-space $\mathcal{H} \triangleq \{x \mid Hx \leq g\}$ is another star with the following characteristics.*

$$\bar{\Theta} = \Theta \cap \mathcal{H} = \langle \bar{c}, \bar{V}, \bar{P}, \bar{l}, \bar{u}\rangle,\ \bar{c} = c,\ \bar{V} = V,\ \bar{P} = P \wedge P',$$
$$P'(\alpha) \triangleq (H \times V_m)\alpha \leq g - H \times c, V_m = [v_1\ v_2 \cdots v_m],$$
$$\bar{l} = l,\ \bar{u} = u.$$

Figure 3.3 shows an example of the intersection between a star set and a half-space.
The following code demonstrates constraint operations on star sets. Constraints can be added in three equivalent ways:

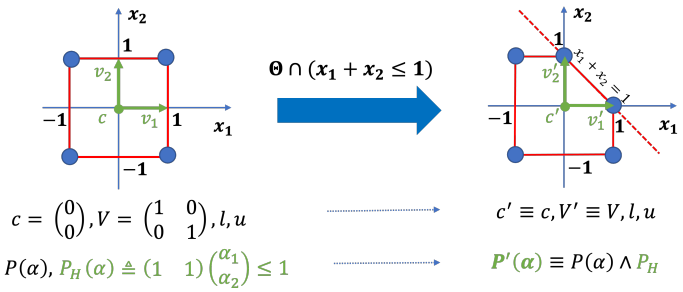
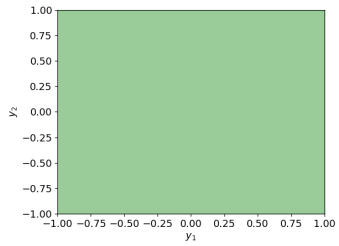


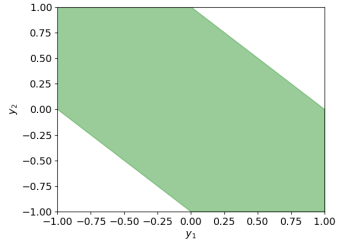
Figure 3.3: Intersection of a star set with a half-space.

```python
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
S = Star(lb, ub)

print("\nOriginal star set:")
print(S)
plot_star(S)

# 1. Adding single constraints one at a time:
S1 = copy.deepcopy(S)
# Add the first constraint:
# alpha_1 + alpha_2 <= 1
C1 = np.array([1.0, 1.0])
d1 = np.array([1])
S1.addConstraint(C1, d1)
# Add the second constraint:
# -alpha_1 - alpha_2 <= 1
C2 = np.array([-1.0, -1.0])
d2 = np.array([1])
S1.addConstraint(C2, d2)

# 2. Adding multiple constraints at once:
S2 = copy.deepcopy(S)
# Combined constraints
C = np.array([[1.0, 1.0],
              [-1.0, -1.0]])
d = np.array([1, 1])
S2.addMultipleConstraints(C, d)

S3 = copy.deepcopy(star)
S3 = S3.intersectHalfSpace(C, d)

print("\nStar set after adding constraints:")
print(S3)
plot_star(S3)
```

```
Original star set:
# print(S)
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]

Star set after adding constraints:
# print(S3)
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[-1. -1.]
 [ 1.  1.]]
d: [1. 1.]
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
```



plot_star(S)



plot_star(S3)

1.6 Set operations

Proposition 6. *[Minkowski sum of two stars] Given two stars* $\Theta_1 = \langle c_1, V_1, P_1, l_1, u_1 \rangle$ *and* $\Theta_2 = \langle c_2, V_2, P_2, l_2, u_2 \rangle$ *with the same dimensions, the Minkowski sum of the two stars is another star set* $\Theta = \Theta_1 \oplus \Theta_2 = \langle c, V, P, l, u \rangle$*, where* $c = c_1 + c_2$*,* $V = [V_1 \; V_2]$*,* $P = P_1 \wedge P_2$*,* $l = [l_1 \; l_2]^T$*, and* $u = [u_1 \; u_2]^T$*.*

The following code demonstrates two key set operations on star sets. 1) **Empty Set Checking:** A star set is empty if its predicate $P(\alpha)$ has no feasible solution. 2) **Minkowski Sum:** For two stars Θ_1 and Θ_2, their Minkowski sum is computed as described above:

```python
# 1. Empty Set Checking:
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
S = Star(lb, ub)
```

```
1.
Is original star set empty? False
Is the new constrained star set empty? True

2.
# print(S1)
```

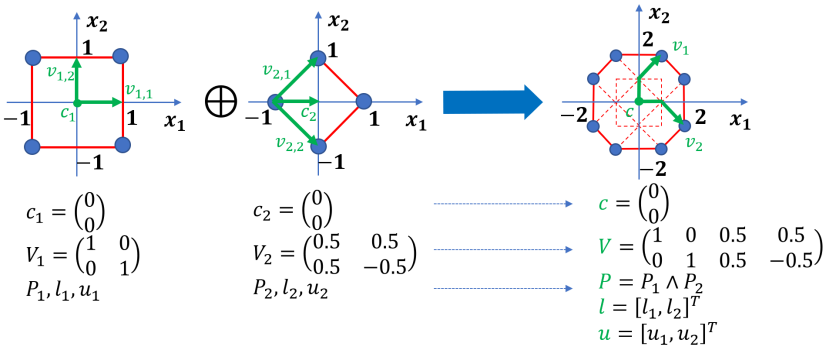
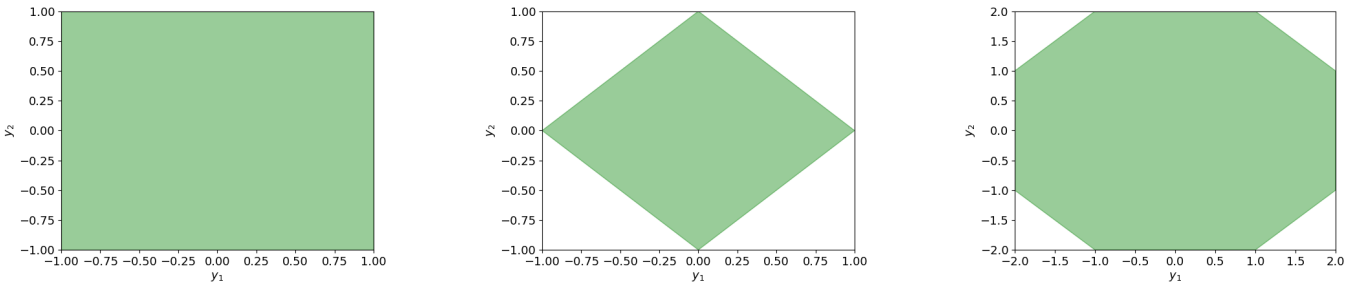


Figure 3.4: Minkowski Sum of two star sets.

```
5
6   empty = S.isEmptySet()
7   print(f"\nIs original star set empty? {empty}")
8
9   # Create an empty star set by adding an infeasible constraint:
10  # Constraint: -alpha_1 <= -2 (infeasible with alpha_1 <= 1)
11  C = np.array([-1.0, 0.0])
12  d = np.array([-2])
13  S.addConstraint(C, d)
14
15  empty_constrained = S.isEmptySet()
16  print(f"Is the new constrained star set empty?
    ↪  {empty_constrained}")
17
18  # 2. Minkowski Sum:
19  lb = np.array([-1.0, -1.0])
20  ub = np.array([1.0, 1.0])
21  S1 = Star(lb, ub)
22
23  print(S1)
24  plot_star(S1)
25
26  W = np.array([[0.5, 0.5],
27                [0.5, -0.5]])
28  b = np.array([0, 0])
29  S2 = S1.affineMap(W, b)
30
31  print(S2)
32  plot_star(S2)
33
34  # Compute the Minkowski sum:
35  S_sum = S1.minKowskiSum(S2)
36
37  print(S_sum)
38  plot_star(S_sum)
```

```
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]

# print(S2)
Star Set:
V: [[ 0.   0.5  0.5]
 [ 0.   0.5 -0.5]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]

print(S_sum)
Star Set:
V: [[ 0.   1.   0.   0.5  0.5]
 [ 0.   0.   1.   0.5 -0.5]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 4
pred_lb: [-1. -1. -1. -1.]
pred_ub: [1. 1. 1. 1.]
```



(a) `plot_star(S1)`. (b) `plot_star(S2)`. (c) `plot_star(S_sum)`.

Figure 3.5: Example of Minkowski Sum of two star sets.

2 Probabilistic Star (ProbStar)

2.1 ProbStar set construction

In StarV, we support multiple ways of constructing a probstar set (Def. 2). Some examples are provided below.

Definition 2 (Probabilistic Star [9]). *A probabilistic star (or simply ProbStar) Θ is a tuple $\langle c, V, \mathcal{N}, P, l, u\rangle$ 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, $P : \mathbb{R}^m \to \{\top, \bot\}$ is a predicate, l and u are the lower-bound and upper-bound vectors of the predicate variables, which are random variables of a Gaussian distribution $\mathcal{N}$. The basis vectors are arranged to form the ProbStar's $n \times m$ basis matrix. The set of states represented by the ProbStar is given as:*

$$[\![\Theta]\!] = \{x \mid x = c + \Sigma_{i=1}^{m}(\alpha_i v_i),\ \alpha = [\alpha_1, \cdots, \alpha_m]^T \sim \mathcal{N}, P(\alpha) \triangleq C\alpha \leq d,\ l[i] \leq \alpha_i \leq u[i]\}. \tag{3.1}$$

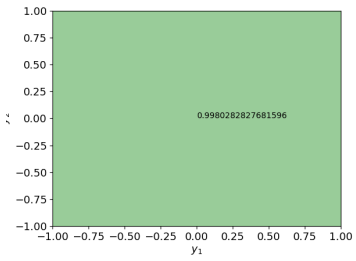
We will refer to both the tuple Θ and the set of states $[\![\Theta]\!]$ as Θ.

Method 1: Construction of ProbStar using bounded Gaussian distribution

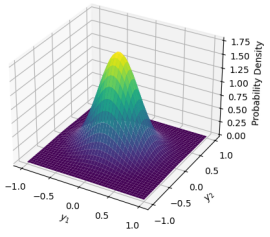
In this method, the ProbStar set is constructed by bounded Gaussian distribution, i.e., $x \sim \mathcal{N}(\mu, \Sigma)$.

```
dim = 2

mu = np.zeros(dim)
Sig = np.diag([0.1, 0.08])
pred_lb = -np.ones(dim)
pred_ub = np.ones(dim)

P = ProbStar(mu, Sig, pred_lb, pred_ub)

print(P)
plot_probstar(P)
plot_probstar_distribution(P)
plot_probstar_contour(P)

```

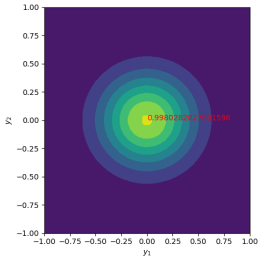
```
ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
mu: [0. 0.]
Sig: [[0.1  0.  ]
 [0.   0.08]]
```



(a) `plot_probstar(P)`.



(b) `plot_probstar_distribution(P)`.



(c) `plot_probstar_contour(P)`.

Figure 3.6: Construction using bounded Gaussian distribution.

Method 2: Construction of ProbStar set

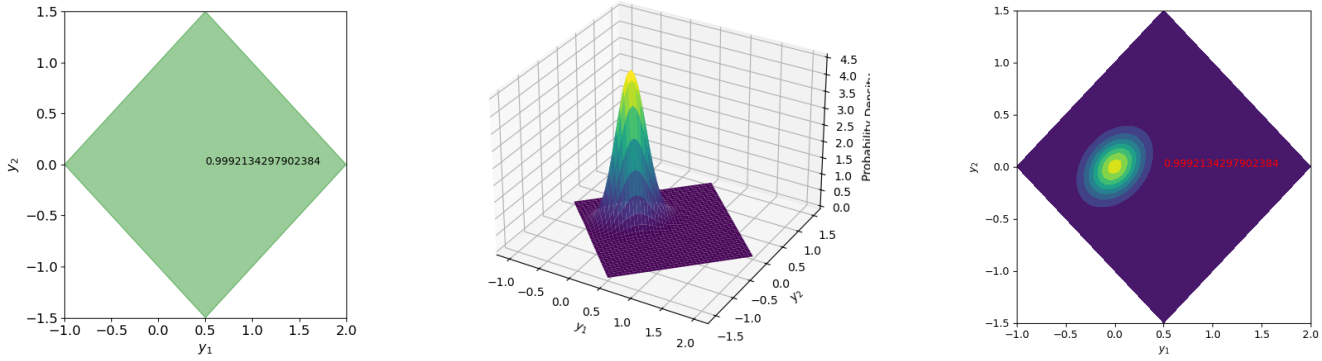
In this method, the ProbStar set is constructed by explicitly specifying the center vector, the basis matrix, the linear constraints of the predicate, and the Gaussian distribution of the predicate variables.

```
dim = 2

center = np.zeros([dim, 1])
basis_matrix = np.array([[0.5, 0.5], [-0.5, 0.5]])
V = np.concatenate([center, basis_matrix], axis=1)

C = np.concatenate([np.eye(dim), -np.eye(dim)], axis=0)
d = np.array([2, 2, 1, 1])
```

```
ProbStar Set:
V: [[ 0.   0.5  0.5]
 [ 0.  -0.5  0.5]]
Predicate Constraints:
C: [[ 1.  0.]
 [ 0.  1.]
 [-1. -0.]
 [-0. -1.]]
d: [2 2 1 1]
dim: 2
nVars: 2
```

```
mu = np.zeros(dim)
Sig = np.diag([0.05, 0.1])
pred_lb = -np.ones(dim)
pred_ub = 2*np.ones(dim)

P = ProbStar(V, C, d, mu, Sig, pred_lb, pred_ub)

print(P)
plot_probstar(P)
plot_probstar_distribution(P)
plot_probstar_contour(P)
```

```
pred_lb: [-1. -1.]
pred_ub: [2. 2.]
mu: [0. 0.]
Sig: [[0.05 0.  ]
 [0.   0.1 ]]
```



(a) `plot_probstar(P)`. (b) `plot_probstar_distribution(P)`. (c) `plot_probstar_contour(P)`.

Figure 3.7: Construction ProbStar set

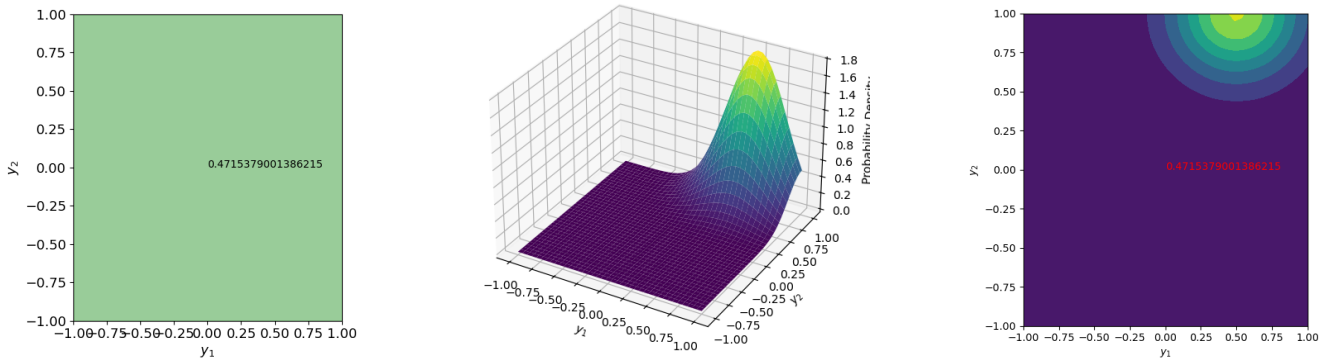
2.2 Affine mapping of ProbStar

Proposition 7 (Affine Mapping)**.** *Given a ProbStar set* $\Theta = \langle c, V, \mathcal{N}, P, l, u\rangle$*, an affine mapping of the ProbStar* Θ *with the mapping matrix* W *and offset vector* b *defined by* $\bar{\Theta} = \{y \mid y = Wx + b,\ x \in \Theta\}$ *is another probstar with the following characteristics:* $\bar{\Theta} = \langle \bar{c}, \bar{V}, \bar{\mathcal{N}}, \bar{P}, \bar{l}, \bar{u}\rangle$*,* $\bar{c} = Wc + b$*,* $\bar{v} = \{Wv_1, Wv_2, \cdots, Wv_m\}$*,* $\bar{\mathcal{N}} = \mathcal{N}$*,* $\bar{P} \equiv P, \bar{l} \equiv l, \bar{u} \equiv u$*.*

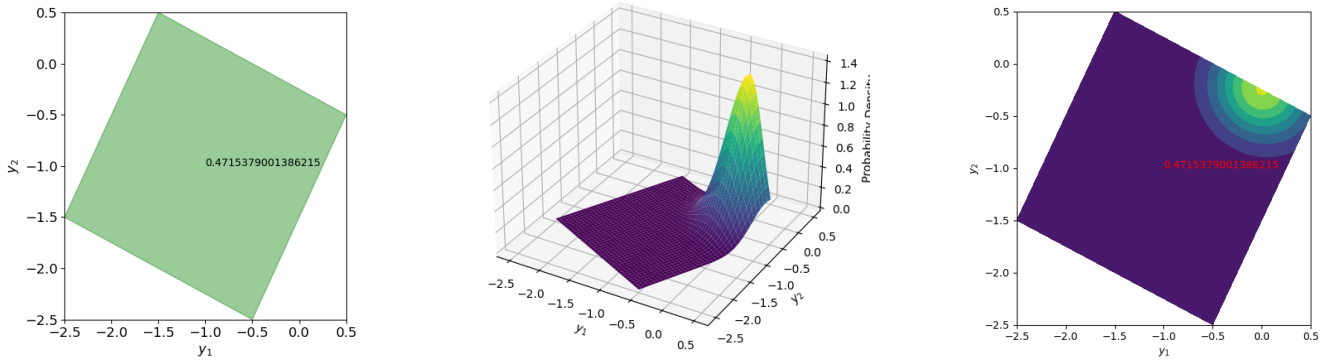
The following example illustrates the affine mapping operation of ProbStar. The ProbStar is constructed based on the bounded Gaussian distribution and then affined transformed by a transformation matrix A and a translation vector b.

```
dim = 2

# original ProbStar
mu = np.array([0.5, 1.0])
Sig = np.diag([0.1, 0.08])
pred_lb = -np.ones(dim)
pred_ub = np.ones(dim)

P = ProbStar(mu, Sig, pred_lb, pred_ub)
print(P)
plot_probstar(P)
plot_probstar_distribution(P)
plot_probstar_contour(P)

# affine mapped ProbStar
A = np.array([[1.0, 0.5], [-0.5, 1.0]])
b = -np.ones(dim)
P1 = P.affineMap(A, b)

print(P1)
plot_probstar(P1)
plot_probstar_distribution(P1)
plot_probstar_contour(P1)
```

```
# original ProbStar
ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
mu: [0.5 1. ]
Sig: [[0.1  0.  ]
 [0.   0.08]]

# affine mapped ProbStar
ProbStar Set:
V: [[-1.   1.   0.5]
 [-1.  -0.5  1. ]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
mu: [0.5 1. ]
Sig: [[0.1  0.  ]
 [0.   0.08]]
```



(a) `plot_probstar(P)`. (b) `plot_probstar_distribution(P)`. (c) `plot_probstar_contour(P)`.



(d) `plot_probstar(P1)`. (e) `plot_probstar_distribution(P1)`. (f) `plot_probstar_contour(P1)`.

Figure 3.8: Affine mapping of ProbStar

2.3 Compute state bounds of ProbStar

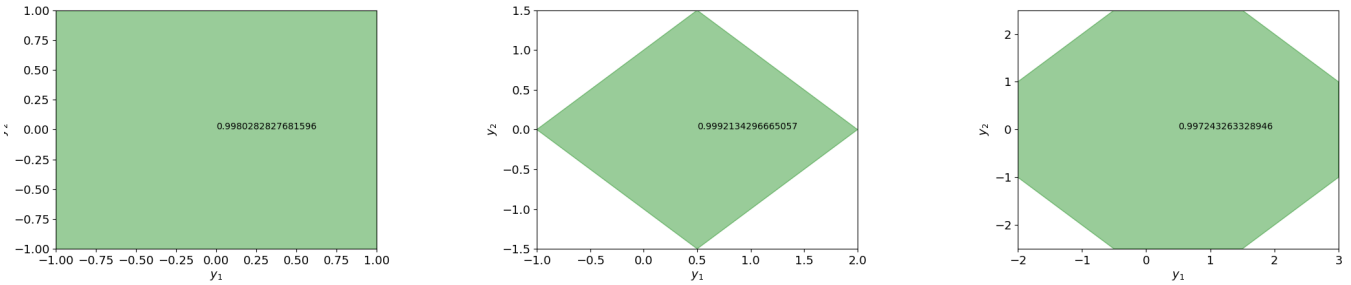
```
dim = 2

center = np.zeros([dim, 1])
basis_matrix = np.array([[1.0, 0.5], [-0.5, 1.0]])
V = np.concatenate([center, basis_matrix], axis=1)

C = np.concatenate([np.eye(dim), -np.eye(dim)], axis=0)
d = np.array([2, 2, 1, 1])

mu = np.zeros(dim)
Sig = np.diag([0.05, 0.1])
pred_lb = -np.ones(dim)
pred_ub = 2*np.ones(dim)

# Create ProbStar set
P = ProbStar(V, C, d, mu, Sig, pred_lb, pred_ub)

# Estimated range for {index} dimension
index = 0
l_est_0, u_est_0 = P.estimateRange(index)

# Estimate ranges for all dimensions
l_est, u_est = P.estimateRanges()

# Estimate ranges for all dimensions
x_mins, x_maxs = P.getRanges()
```

```
Estimated range for 0 dimension:
Estimated 0th lower state bound (l_est_0) = [-1.5]
Estimated 0th upper state bound (u_est_0) = [3.]

Estimated ranges for all dimensions:
Estimated lower state bounds (l_est) = [-1.5 -2. ]
Estimated upper state bounds (u_est) = [3.  2.5]

Optimized ranges for all dimensions:
Optimized lower state bounds (x_mins) = [-1.5 -2. ]
Optimized upper state bounds (x_maxs) = [3.  2.5]
```

2.4 Minkowski sum of two ProbStar sets

Currently, `plot_probstar_distribution` and `plot_probstar_contour` are not supported after the Minkowski Sum of two ProbStar sets.

```
dim = 2
mu = np.zeros(dim)
Sig = np.diag([0.1, 0.08])
pred_lb = -np.ones(dim)
pred_ub = np.ones(dim)

# Create first ProbStar set
P1 = ProbStar(mu, Sig, pred_lb, pred_ub)

center = np.zeros([dim, 1])
basis_matrix = np.array([[0.5, 0.5], [-0.5, 0.5]])
V = np.concatenate([center, basis_matrix], axis=1)

C = []
d = []

mu = np.zeros(dim)
Sig = np.diag([0.05, 0.1])
pred_lb = -np.ones(dim)
pred_ub = 2*np.ones(dim)

# Create second ProbStar set
P2 = ProbStar(V, C, d, mu, Sig, pred_lb, pred_ub)

# Minkowski sum of two ProbStar sets
P = P1.minKowskiSum(P2)
```



(a) First ProbStar set. (b) Second ProbStar set. (c) Minkowski Sum of two ProbStars.

Figure 3.9: Example of Minkowski Sum of two ProbStar sets.

3 Sparse Star (SparseStar)

Definition 3 (Sparse Star Set)**.** A SparseStar set *(or simply SPstar)* Φ *is a tuple* $\langle c, B, P, \tau \rangle$ *where* $c \in \mathbb{R}^n$ *is the center,* $B = \{b_1, b_2, ..., b_{m-k}\}$ *is a set of* $m-k$ *independent basis vectors in* $\mathbb{R}^n, P : \mathbb{R}^m \to \{\top, \bot\}$ *is a predicate,* $\tau \in (\mathbb{Z}^+)^m$ *is the depth of predicate variables. The independent basis vectors are arranged to form the sparse star's* $n \times m-k$ *independent basis matrix. The basis matrix can be reformulated by* $V = [Z,\ B]$*, where* $Z = \mathbf{0}^{n \times k}$*. The set of states represented by the sparse star is given as:*

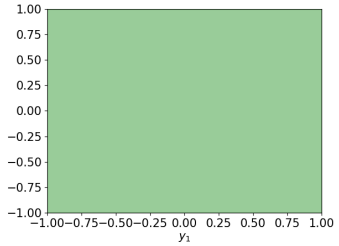
$$[\![\Phi]\!] = \{x \mid x = c + V\alpha,\ P(\alpha_1, \cdots, \alpha_m) = \top\}.$$

For the sake of brevity, we refer to both the tuple Φ *and the set of states* $[\![\Phi]\!]$ *as* Φ*. In this work, we restrict the predicates to be a conjunction of linear constraints,* $P(\alpha) \triangleq C\alpha \leq d \wedge l \leq \alpha \leq u$ *where, for* p *linear constraints,* $C \in \mathbb{R}^{p \times m}$ *is a compressed sparse column (CSC) matrix,* α *is the vector of* m *predicate variables, i.e.,* $\alpha = [\alpha_1, \cdots, \alpha_m]^T$*,* $d \in \mathbb{R}^{p \times 1}$*, and* $l, u \in \mathbb{R}^m$ *are the lower-bound and upper-bound vectors of the predicate variables. A sparse star is an empty set if and only if* $P(\alpha)$ *is empty.*

Method 1: Construction of SparseStar using state bounds

```
dim = 2

# Define bounds
lb = -np.ones(dim)       # lower bounds: x1 >= -1, x2 >= -1
ub = np.ones(dim)        # upper bounds: x1 <= 1, x2 <= 1

# Create star set
S = SparseStar(lb, ub)

print("Created box-shaped SparseStar set:")
print(S)
plot_star(S)
```

```
Created box-shaped SparseStar set:
SparseStar Set:
A:
[[0. 1. 0.]
 [0. 0. 1.]]
C_csc:
[]
d: []
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
pred_depth: [0. 0.]
dim: 2
nVars: 2
nZVars: 0
nIVars: 2
```

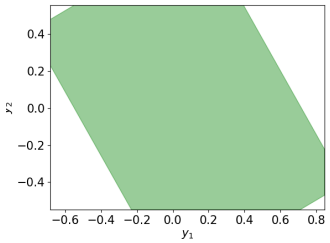


plot_star(S)

Method 2: Construction of SparseStar

```
dim = 2

c = np.zeros([2, 1])
v = np.eye(dim)
A = np.hstack([c, v])

C = np.array([
    [-0.22647,  0.06832, -1,  0],
    [ 0.39032,  0.03921,  0, -1],
    [ 0.22647, -0.06832,  1,  0],
    [-0.39032, -0.03921,  0,  1]])
C = sp.csc_array(C)
d = np.array([0.3890, 0.1199, 0.5516, 0.1285])
pred_lb = np.array([-1, -1, -0.68382, -0.54943])
pred_ub = np.array([ 1,  1,  0.84647, 0.55803])
pred_depth = np.array([1, 1, 0, 0])
S = SparseStar(A, C, d, pred_lb, pred_ub, pred_depth)

print("Created SparseStar set with independent basis matrix and constraints:")
repr(S)
print(S)
plot_star(S)
```

```
Created SparseStar set with independent basis matrix and constraints:

#repr(S)
SparseStar Set:
A: (2, 3)
C_csc: (4, 4)
d: (4,)
pred_lb: (4,)
pred_ub: (4,)
pred_depth: (4,)
dim: 2
nVars: 4
nZVars: 2
nIVars: 2

#print(S)
SparseStar Set:
A:
[[0. 1. 0.]
 [0. 0. 1.]]
C_csc:
[[-0.22647  0.06832 -1.       0.     ]
 [ 0.39032  0.03921  0.      -1.     ]
 [ 0.22647 -0.06832  1.       0.     ]
 [-0.39032 -0.03921  0.       1.     ]]
d: [0.389  0.1199 0.5516 0.1285]
pred_lb: [-1.      -1.      -0.68382
 -0.54943]
pred_ub: [1.      1.      0.84647 0.55803]
pred_depth: [1 1 0 0]
dim: 2
nVars: 4
nZVars: 2
nIVars: 2
```

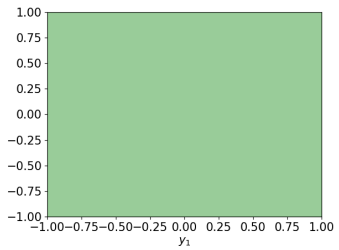


plot_star(S)

Method 3: Affine mapping of SparseStar

```
dim = 2

# original SparseStar
lb = -np.ones(dim)  # lower bounds: x1 >= -1, x2 >= -1
ub = np.ones(dim)   # upper bounds: x1 <= 1, x2 <= 1
S = SparseStar(lb, ub)
plot_star(S)

# affine mapped SparseStar
A = np.array([[1.0, 0.5], [-0.5, 1.0]])
b = -np.ones(dim)
S1 = S.affineMap(A, b)

```

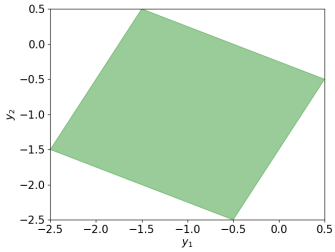
```
Original SparseStar set:
SparseStar Set:
A:
[[0. 1. 0.]
 [0. 0. 1.]]
C_csc:
[]
d: []
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
pred_depth: [0. 0.]
dim: 2
nVars: 2
nZVars: 0
nIVars: 2

SparseStar set after affine mapping:
SparseStar Set:
A:
[[-1.   1.   0.5]
 [-1.  -0.5  1. ]]
```



plot_star(S)

```
print(S)
print(S1)
plot_star(S1)
```

```
C_csc:
[]
d: []
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
pred_depth: [0. 0.]
dim: 2
nVars: 2
nZVars: 0
nIVars: 2
```

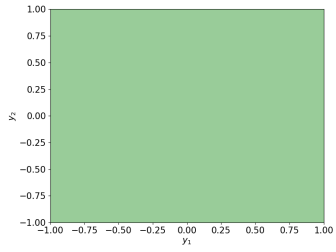


plot_star(S1)

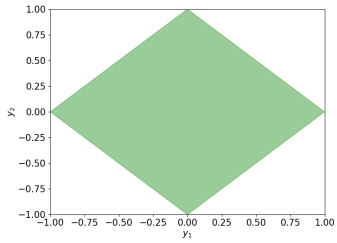
Method 4: Minkowsi sum of SparseStar

```
dim = 2

# Create first SparseStar set
lb = -np.ones(dim)
ub = np.ones(dim)
S1 = SparseStar(lb, ub)

print("SparseStar set 1:")
print(S1)
plot_star(S1)

# Create second SparseStar set using affine map of first set
W = np.array([[0.5, 0.5],
              [0.5, -0.5]])
b = np.array([0, 0])
S2 = S1.affineMap(W, b)

print("SparseStar set 2:")
print(S2)
plot_star(S2)

# Compute Minkowski sum
S = S1.minKowskiSum(S2)

print("SparseStar after Minkowski sum:")
print(S)
plot_star(S)
```

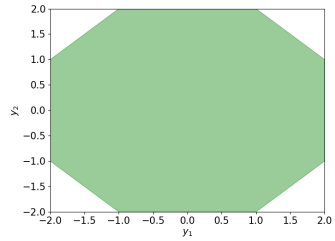
```
SparseStar set 1:
SparseStar Set:
A:
[[0. 1. 0.]
 [0. 0. 1.]]
C_csc:
[]
d: []
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
pred_depth: [0. 0.]
dim: 2
nVars: 2
nZVars: 0
nIVars: 2

SparseStar set 2:
SparseStar Set:
A:
[[ 0.   0.5  0.5]
 [ 0.   0.5 -0.5]]
C_csc:
[]
d: []
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
pred_depth: [0. 0.]
dim: 2
nVars: 2
nZVars: 0
nIVars: 2

SparseStar after Minkowski sum:
SparseStar Set:
A:
[[ 0.   1.   0.   0.5  0.5]
 [ 0.   0.   1.   0.5 -0.5]]
C_csc:
[]
d: []
pred_lb: [-1. -1. -1. -1.]
pred_ub: [1. 1. 1. 1.]
pred_depth: [0. 0. 0. 0.]
dim: 2
nVars: 4
nZVars: 0
nIVars: 4
```



plot_star(S1)



plot_star(S2)

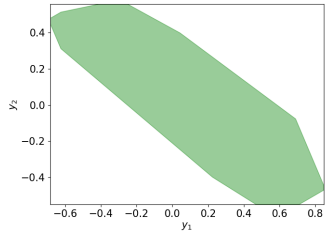


plot_star(S)

Method 5: Predicate reduction of SparseStar by predicate depth

```
dim = 2

c = np.zeros([2, 1])
v = np.eye(dim)
A = np.hstack([c, v])

C = np.array([
    [-0.22647,  0.06832, -1,  0],
    [ 0.39032,  0.03921,  0, -1],
    [ 0.22647, -0.06832,  1,  0],
    [-0.39032, -0.03921,  0,  1],
    [-0.62899,  0.18975, -1,  0],
    [ 0.52719,  0.05296,  0, -1],
```

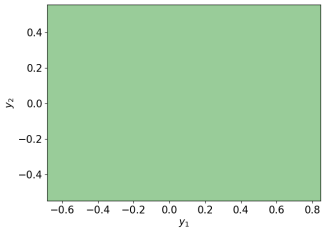
```
Original SparseStar:
SparseStar Set:
A:
[[0. 1. 0.]
 [0. 0. 1.]]
C_csc:
[[-0.22647  0.06832 -1.       0.     ]
 [ 0.39032  0.03921  0.      -1.     ]
 [ 0.22647 -0.06832  1.       0.     ]
 [-0.39032 -0.03921  0.       1.     ]
 [-0.62899  0.18975 -1.       0.     ]
 [ 0.52719  0.05296  0.      -1.     ]
 [ 0.71908 -0.21693  1.       0.     ]
 [-0.52883 -0.05312  0.       1.     ]]
d: [ 0.389    0.1199   0.5516   0.1285
↪  -0.02773  0.02212  0.25219  0.03252]
pred_lb: [-1.      -1.      -0.68382
↪  -0.54943]
```



plot_star(S)

```
    [ 0.71908, -0.21693,  1,  0],
    [-0.52883, -0.05312,  0,  1]])
C = sp.csc_array(C)
d = np.array([0.3890, 0.1199, 0.5516,
↪  0.1285, -0.02773, 0.02212, 0.25219,
↪  0.03252])
pred_lb = np.array([-1, -1, -0.68382,
↪  -0.54943])
pred_ub = np.array([ 1,  1,  0.84647,
↪  0.55803])
pred_depth = np.array([1, 1, 0, 0])
S = SparseStar(A, C, d, pred_lb, pred_ub,
↪  pred_depth)

print('Original SparseStar:')
print(S)
plot_star(S)

print('Predicate depth reduced SparseStar
↪  with DR=1:')
D = S.depthReduction(DR=1)
print(D)
plot_star(D)
```

```
pred_ub: [1.      1.      0.84647 0.55803]
pred_depth: [1 1 0 0]
dim: 2
nVars: 4
nZVars: 2
nIVars: 2

Predicate depth reduced SparseStar with
↪  DR=1:
SparseStar Set:
A:
[[0. 1. 0.]
 [0. 0. 1.]]
C_csc:
[]
d: []
pred_lb: [-0.68382 -0.54943]
pred_ub: [0.84647 0.55803]
pred_depth: [0 0]
dim: 2
nVars: 2
nZVars: 0
nIVars: 2
```

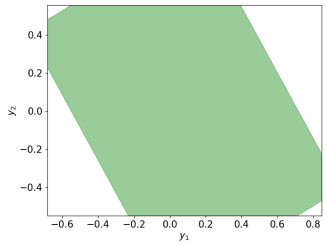


plot_star(D)

Method 6: Compute state bounds of SparseStar

```
dim = 2

c = np.zeros([2, 1])
v = np.eye(dim)
A = np.hstack([c, v])

C = np.array([
    [-0.22647,  0.06832, -1,  0],
    [ 0.39032,  0.03921,  0, -1],
    [ 0.22647, -0.06832,  1,  0],
    [-0.39032, -0.03921,  0,  1]
])
C = sp.csc_array(C)
d = np.array([0.3890, 0.1199, 0.5516,
↪  0.1285])
pred_lb = np.array([-1, -1, -0.68382,
↪  -0.54943])
pred_ub = np.array([ 1,  1,  0.84647,
↪  0.55803])
pred_depth = np.array([1, 1, 0, 0])
S = SparseStar(A, C, d, pred_lb, pred_ub,
↪  pred_depth)
print('Constructed SparseStar:')
print(S)
plot_star(S)

l, u = S.getRanges()
print('State bounds computed with
↪  getRanges() via LP solver:')
print('lower bounds:\n', l)
print('upper bounds:\n', u)
print()

l, u = S.estimateRanges()
print('State bounds computed with
↪  estimateRanges():')
print('lower bounds:\n', l)
print('upper bounds:\n', u)
print()

l, u = S.getRanges('estimate')
print('State bounds computed with
↪  getRanges(''estimate'')')
```

```
Constructed SparseStar:
SparseStar Set:
A:
[[0. 1. 0.]
 [0. 0. 1.]]
C_csc:
[[-0.22647  0.06832 -1.       0.     ]
 [ 0.39032  0.03921  0.      -1.     ]
 [ 0.22647 -0.06832  1.       0.     ]
 [-0.39032 -0.03921  0.       1.     ]]
d: [0.389  0.1199 0.5516 0.1285]
pred_lb: [-1.      -1.      -0.68382
↪  -0.54943]
pred_ub: [1.      1.      0.84647 0.55803]
pred_depth: [1 1 0 0]
dim: 2
nVars: 4
nZVars: 2
nIVars: 2

State bounds computed with getRanges() via
↪  LP solver:
lower bounds:
 [-0.68379 -0.54943]
upper bounds:
 [0.84639 0.55803]

State bounds computed with estimateRanges():
lower bounds:
 [-0.68382 -0.54943]
upper bounds:
 [0.84647 0.55803]

State bounds computed with
↪  getRanges(estimate)
lower bounds:
 [-0.68382 -0.54943]
upper bounds:
 [0.84647 0.55803]
```



plot_star(S)

```
print('lower bounds:\n', l)
print('upper bounds:\n', u)
print()
```

4 Image Star (ImageStar)

4.1 ImageStar set construction

Definition 4 (ImageStar). *An ImageStar Θ is a tuple $\langle c, V, P\rangle$, where $c \in \mathbb{R}^{h\times w\times ch}$ is the anchor image, $V = \{v_1, v_2, \cdots, v_m\}$ is a set of m images in $\mathbb{R}^{h\times w\times ch}$ called generator images, $P : \mathbb{R}^m \to \{\top, \bot\}$, is a predicate, and h, w, ch are the height, width, and number of channels of the images respectively. The generator images are arranged to form the ImageStar's $h \times w \times ch \times m$ basis array. The set of images represented by the ImagesStar is given as:*

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

Sometimes, we will refer to both the tuple Θ and the set of state $[\![\Theta]\!]$ as Θ. In this work, we restrict the predicates to be a conjunction of linear constraints, $P(\alpha) \triangleq C\alpha \leq d$ where, for p linear constraints, $C \in \mathbb{R}^{p\times m}, \alpha$ is the vector of m-variables, i.e., $\alpha = [\alpha_1, \cdots, \alpha_m]^T$, and $d \in \mathbb{R}^{p\times 1}$. An ImageStar is an empty set if and only if $P(\alpha)$ is empty.

Example 1 (ImageStar). *A $4 \times 4 \times 1$ image with a bounded disturbance $b \in [-2, 2]$ applied on the pixel of the position $(1, 2, 1)$ can be described as an ImageStar depicted in Figure 3.10.*

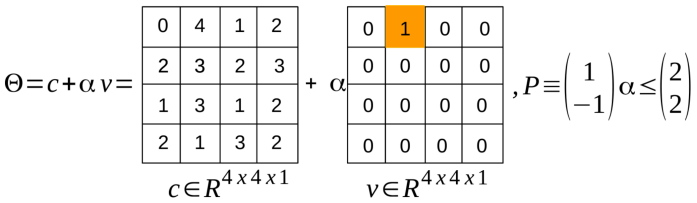


Figure 3.10: An example of an ImageStar set.

Method 1: Construction of ImageStar using state bounds

```
c = np.array([[2, 3], [8, 0]])
a = np.zeros([2, 2])
a[0, 0] = 2
a[1, 0] = 2
a[0, 1] = 2
lb = c - a
ub = c + a
IM = ImageStar(lb, ub)
print(IM)
repr(IM)
```

```
# print(IM)
ImageStar Set:
V:
[[[[2. 2. 0. 0.]]
  [[3. 0. 2. 0.]]]
 [[[8. 0. 0. 2.]]
  [[0. 0. 0. 0.]]]]
C:
[]
d: []
pred_lb: [-1. -1. -1.]
pred_ub: [1. 1. 1.]
height: 2
width: 2
num_channel: 1
num_pred: 3
```

```
# repr(IM)
ImageStar Set:
V: (2, 2, 1, 4), float64
C: (0, 0), float64
d: (0,), float64
pred_lb: (3,), float64
pred_ub: (3,), float64
height: 2
width: 2
num_channel: 1
num_pred: 3
```

Method 2: Construction of ImageStar

```
c = np.array([[0,4,1,2], [2,3,2,3],
↪  [1,3,1,2], [2,1,3,2]])[:,:,None,None]
V = np.zeros([4, 4, 1, 1])
V[0, 1, 0, 0] = 1
V = np.concatenate([c, V], axis=3)

# predicate constraints
C = np.zeros([2, 1])
d = np.zeros(2)
# 1 a <= 2
C[0, 0] = 1
```

```
# print(IM)
ImageStar Set:
V:
[[[[0. 0.]]
  [[4. 1.]]
  [[1. 0.]]
  [[2. 0.]]]
 [[[2. 0.]]
  [[3. 0.]]
  [[2. 0.]]
  [[3. 0.]]]
 [[[1. 0.]]
  [[3. 0.]]
  [[1. 0.]]
  [[2. 0.]]]
```

```
# repr(IM)
ImageStar Set:
V: (4, 4, 1, 2), float64
C: (2, 1), float64
d: (2,), float64
pred_lb: (1,), float64
pred_ub: (1,), float64
height: 4
width: 4
num_channel: 1
num_pred: 1
```

```
d[0] = 2
# -1 a <= 2
C[1, 0] = -1
d[1] = 2

# predicate bounds
# -2 <= a <= 2
pred_lb = -2*np.ones(1)
pred_ub = 2*np.ones(1)

IM = ImageStar(V, C, d, pred_lb, pred_ub)
print(IM)
repr(IM)
```

```
 [[[2. 0.]]
  [[1. 0.]]
  [[3. 0.]]
  [[2. 0.]]]]
C:
[[ 1.]
 [-1.]]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
height: 4
width: 4
num_channel: 1
num_pred: 1
```

4.2 Affine Mapping of ImageStar

Proposition 8 (Affine mapping of an ImageStar)**.** *An affine mapping of an ImageStar $\Theta\langle c, V, P\rangle$ with a scale factor γ and an offset image β is another ImageStar $\Theta' = \langle c', V', P'\rangle$ in which the new anchor, generators, and predicate are as follows:*

$$c' = \gamma \times c + \beta, V' = \gamma \times V, P' \equiv P.$$

```
shape = (2, 2, 1)
dim = np.prod(shape)
data = np.arange(dim).reshape(shape) / dim
eps = 0.1
lb = np.clip(data - eps, 0, 1)
ub = np.clip(data + eps, 0, 1)
IM = ImageStar(lb, ub)
print('original ImageStar: \n')
print(IM)
print()

# Apply affine mapping operation
W = np.array([[[-0.3463], [-0.5628]], [[ 0.5825], [ 0.6715]]])
b = np.array([[[ 0.3383], [-0.2682]], [[-0.5748], [-0.3584]]])

# Affine mapped ImageStar
R = IM.affineMap(W, b)

print(f'affine mapping matrix: \n{W}')
print(f'affine mapping bias: \n{b}\n')

print('affine mapped ImageStar:')
print(R)
```

```
original ImageStar:

ImageStar Set:
V:
[[[[0.05 0.05 0.   0.   0.  ]]
  [[0.25 0.   0.1  0.   0.  ]]]
 [[[0.5  0.   0.   0.1  0.  ]]
  [[0.75 0.   0.   0.   0.1 ]]]]
C:
[]
d: []
pred_lb: [-1. -1. -1. -1.]
pred_ub: [1. 1. 1. 1.]
height: 2
width: 2
num_channel: 1
num_pred: 4

affine mapping matrix:
[[[-0.3463]
  [-0.5628]]
 [[ 0.5825]
  [ 0.6715]]]
affine mapping bias:
[[[ 0.3383]
  [-0.2682]]
 [[-0.5748]
  [-0.3584]]]

affine mapped ImageStar:
ImageStar Set:
V:
[[[[ 0.32098 -0.01732 -0.      -0.      -0.     ]]
  [[-0.4089  -0.      -0.05628 -0.      -0.     ]]]
 [[[-0.28355  0.       0.       0.05825  0.     ]]
  [[ 0.14522  0.       0.       0.       0.06715]]]]
C:
[]
d: []
pred_lb: [-1. -1. -1. -1.]
pred_ub: [1. 1. 1. 1.]
height: 2
width: 2
num_channel: 1
num_pred: 4
```

4.3 Compute state bounds of ImageStar

```
c = np.array([[2, 3], [8, 0]])
a = np.zeros([2, 2])
a[0, 0] = 2
a[1, 0] = 2
```

```
Actual state bounds of ImageStar:
lower bounds:
 [[0. 1.]
 [6. 0.]]
upper bounds:
 [[ 4.  5.]
 [10.  0.]]
```

```
a[0, 1] = 2
lb = c - a
ub = c + a

IM = ImageStar(lb, ub)
H, W, C = IM.shape
print('Actual state bounds of ImageStar:')
print('lower bounds:\n', lb.reshape(H, W))
print('upper bounds:\n', ub.reshape(H, W))

l, u = IM.getRanges()
print('State bounds computed with getRanges() via LP solver:')
print('lower bounds:\n', l.reshape(H, W))
print('upper bounds:\n', u.reshape(H, W))

l, u = IM.estimateRanges()
print('State bounds computed with estimateRanges():')
print('lower bounds:\n', l.reshape(H, W))
print('upper bounds:\n', u.reshape(H, W))

l, u = IM.getRanges('estimate')
print('State bounds computed with getRanges(\'estimate\')')
print('lower bounds:\n', l.reshape(H, W))
print('upper bounds:\n', u.reshape(H, W))
```

```
State bounds computed with getRanges() via LP solver:
lower bounds:
 [[0. 1.]
 [6. 0.]]
upper bounds:
 [[ 4.  5.]
 [10.  0.]]

State bounds computed with estimateRanges():
lower bounds:
 [[0. 1.]
 [6. 0.]]
upper bounds:
 [[ 4.  5.]
 [10.  0.]]

State bounds computed with getRanges('estimate'):
lower bounds:
 [[0. 1.]
 [6. 0.]]
upper bounds:
 [[ 4.  5.]
 [10.  0.]]
```

5 Sparse Image Star (SparseImageStar)

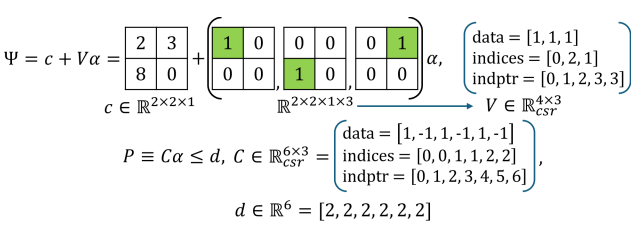
5.1 SparseImageStar set construction

Definition 5 (SparseImageStar). *A SparseImageStar set Ψ is a tuple $\langle c, V, P\rangle$, where $c \in \mathbb{R}^{h\times w\times ch}$ is the center image, $V = \{v_1, v_2, \cdots, v_m\}$ is a set of m flattened images $v \in \mathbb{R}^n$ called generator images, $P : \mathbb{R}^m \to \{\top, \bot\}$ is a predicate, and h, w, ch, n are the height, width, the number of channels of the images, and the number pixels in images respectively. The generator images are arranged to form the SparseImageStar's $n \times m$ sparse matrix in COO or CSR format. The set of images represented by the SparseImageStar is defined as:*

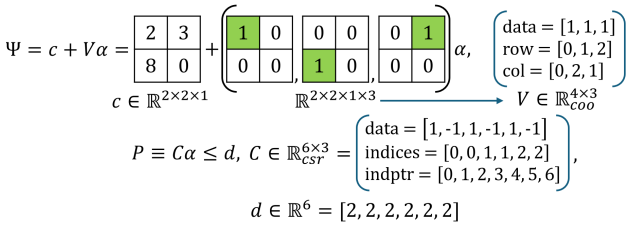
$$[\![\Psi]\!] = \{x \,|\, x = c + \Sigma_{i=1}^{m}(\alpha_i v_i),\ P(\alpha_1, \cdots, \alpha_m) = \top\}.$$

We may refer to both the tuple Ψ and the set of state $[\![\Psi]\!]$ as Ψ. In this work, we restrict the predicates to be a conjunction of linear constraints, i.e. $P(\alpha) \triangleq C\alpha \le d \wedge l \le \alpha \le u$, where $C \in \mathbb{R}^{p\times m}$ is a CSR sparse matrix, $\alpha = [\alpha_1, ..., \alpha_m]$ is m-predicate variables, $d \in \mathbb{R}^p$, l and u are the predicate upper and lower bounds. A SparseImageStar is an infeasible set if $P(\alpha)$ is infeasible. We note that $c_F \in \mathbb{R}^n$ is the flattened center image, where $n = h \times w \times ch$ is the number of pixels in images. All center and generator images, if represented in a 3D array, have the color channel as the last dimension.

Example 2 (SparseImageStar in CSR and COO format). *SparseImageStar sets with CSR and COO format representations of a $2 \times 2 \times 1$ image with bounded disturbances $b \in [-2, 2]$ on the pixel positions $(1, 1)$, $(2, 1)$, and $(1, 2)$ is presented in Fig. 3.11.*



(a) SparseImageStar CSR format



(b) SparseImageStar COO format

Figure 3.11: Examples of a SparseImageStar set representation.

Method 1: Construction of SparseImageStar using state bounds

```
c = np.array([[2, 3], [8, 0]])
a = np.zeros([2, 2])
a[0, 0] = 2
a[1, 0] = 2
a[0, 1] = 2
lb = c - a
ub = c + a

# construction of SparseImageStar COO format
SIM_coo = SparseImageStar2DCOO(lb, ub)
print(SIM_coo)
repr(SIM_coo)

# construction of SparseImageStar COO format
SIM_csr = SparseImageStar2DCSR(lb, ub)
print(SIM_csr)
repr(SIM_csr)
```

```
# print(SIM_coo)
SparseImageStar2DCOO Set:
c: [2. 3. 8. 0.]
V_coo:
  (0, 0)        2.0
  (1, 1)        2.0
  (2, 2)        2.0
C_csr:
   data: []
indices: []
 indptr: [0]
d: []
pred_lb: [-1. -1. -1.]
pred_ub: [1. 1. 1.]
shape: (2, 2, 1)
num_pred: 3
density: 0.25
nnz: 3

# repr(SIM_coo)
SparseImageStar2DCOO Set:
c: (4,), float64
V_coo: (4, 3), data: float64, row: int32, col: int32
C_csr: (0, 0), float64
d: (0,), float64
pred_lb: (3,), float64
pred_ub: (3,), float64
shape: (2, 2, 1)
num_pred: 3
density: 0.25
nnz: 3
```

```
# print(SIM_csr)
SparseImageStar2DCSR Set:
c: [2. 3. 8. 0.]
V_csr:
   data: [2. 2. 2.]
indices: [0 1 2]
 indptr: [0 1 2 3 3]
C:
   data: []
indices: []
 indptr: [0]
d: []
pred_lb: [-1. -1. -1.]
pred_ub: [1. 1. 1.]
shape: (2, 2, 1)
num_pred: 3
density: 0.25
nnz: 3

# repr(SIM_csr)
SparseImageStar2DCSR Set:
c: (4,), float64
V_csr: (4, 3), data: float64, indices: int32, indptr: int32
C_csr: (0, 0), float64
d: (0,), float64
pred_lb: (3,), float64
pred_ub: (3,), float64
shape: (2, 2, 1)
num_pred: 3
density: 0.25
nnz: 3
```

Method 2: Construction of SparseImageStar

```
shape = (2, 2, 1)
# numpred of predicates = number of attacked pixels
num_pred = 3

# center image
c = np.array([2, 3, 8, 0])
# generator image
V = np.zeros(shape + (3,))
V[0, 0, 0, 0] = 1
V[1, 0, 0, 1] = 1
V[0, 1, 0, 2] = 1
V_coo = sp.coo_array(V.reshape(-1, 3))
V_csr = sp.csr_array(V.reshape(-1, 3))

# predicate constraints
E = np.eye(num_pred)
C = np.vstack([E, -E])
C = sp.csr_array(C)
```

```
d = 2*np.ones(2*num_pred)

# predicate bounds
pred_lb = -2*np.ones(num_pred)
pred_ub = 2*np.ones(num_pred)

# construction of SparseImageStar COO format
SIM_coo = SparseImageStar2DCOO(c, V_coo, C, d, pred_lb, pred_ub, shape)
print(SIM_coo)
repr(SIM_coo)

# construction of SparseImageStar COO format
SIM_csr = SparseImageStar2DCSR(c, V_csr, C, d, pred_lb, pred_ub, shape)
print(SIM_csr)
repr(SIM_csr)
```

```
# print(SIM_coo)
SparseImageStar2DCOO Set:
c: [2 3 8 0]
V_coo:
  (0, 0)        1.0
  (1, 2)        1.0
  (2, 1)        1.0
C_csr:
   data: [ 1.  1.  1. -1. -1. -1.]
indices: [0 1 2 0 1 2]
 indptr: [0 1 2 3 4 5 6]
d: [2. 2. 2. 2. 2. 2.]
pred_lb: [-2. -2. -2.]
pred_ub: [2. 2. 2.]
shape: (2, 2, 1)
num_pred: 3
density: 0.25
nnz: 3

# repr(SIM_coo)
SparseImageStar2DCOO Set:
c: (4,), int64
V_coo: (4, 3), data: float64, row: int32, col: int32
C_csr: (6, 3), float64
d: (6,), float64
pred_lb: (3,), float64
pred_ub: (3,), float64
shape: (2, 2, 1)
num_pred: 3
density: 0.25
nnz: 3
```

```
# print(SIM_csr)
SparseImageStar2DCSR Set:
c: [2 3 8 0]
V_csr:
   data: [1. 1. 1.]
indices: [0 2 1]
 indptr: [0 1 2 3 3]
C_csr:
   data: [ 1.  1.  1. -1. -1. -1.]
indices: [0 1 2 0 1 2]
 indptr: [0 1 2 3 4 5 6]
d: [2. 2. 2. 2. 2. 2.]
pred_lb: [-2. -2. -2.]
pred_ub: [2. 2. 2.]
shape: (2, 2, 1)
num_pred: 3
density: 0.25
nnz: 3

# repr(SIM_csr)
SparseImageStar2DCSR Set:
c: (4,), int64
V_csr: (4, 3), data: float64, indices: int32, indptr: int32
C_csr: (6, 3), float64
d: (6,), float64
pred_lb: (3,), float64
pred_ub: (3,), float64
shape: (2, 2, 1)
num_pred: 3
density: 0.25
nnz: 3
```

Method 3: Compute state bounds of SparseImageStar

```
c = np.array([[2, 3], [8, 0]])
a = np.zeros([2, 2])
a[0, 0] = 2
a[1, 0] = 2
a[0, 1] = 2
lb = c - a
ub = c + a

SIM_coo = SparseImageStar2DCOO(lb, ub)
H, W, C = SIM_coo.shape
print('Actual state bounds of SIM_coo:')
print('lower bounds:\n', lb.reshape(H, W))
print('upper bounds:\n', ub.reshape(H, W))

l, u = SIM_coo.getRanges()
print('State bounds computed with getRanges() via LP solver:')
print('lower bounds:\n', l.reshape(H, W))
print('upper bounds:\n', u.reshape(H, W))

l, u = SIM_coo.estimateRanges()
print('State bounds computed with estimateRanges():')
print('lower bounds:\n', l.reshape(H, W))
print('upper bounds:\n', u.reshape(H, W))
```

```
Actual state bounds of SIM_coo:
lower bounds:
 [[0. 1.]
 [6. 0.]]
upper bounds:
 [[ 4.  5.]
 [10.  0.]]

State bounds computed with getRanges() via LP solver:
lower bounds:
 [[0. 1.]
 [6. 0.]]
upper bounds:
 [[ 4.  5.]
 [10.  0.]]

 State bounds computed with estimateRanges():
lower bounds:
 [[0. 1.]
 [6. 0.]]
upper bounds:
 [[ 4.  5.]
 [10.  0.]]

State bounds computed with getRanges('estimate')
lower bounds:
 [[0. 1.]
 [6. 0.]]
upper bounds:
 [[ 4.  5.]
```

```

l, u = SIM_coo.getRanges('estimate')
print('State bounds computed with getRanges(\'estimate\')')
print('lower bounds:\n', l.reshape(H, W))
print('upper bounds:\n', u.reshape(H, W))

SIM_csr = SparseImageStar2DCSR(lb, ub)
H, W, C = SIM_csr.shape
print('Actual state bounds of SIM_csr:')
print('lower bounds:\n', lb.reshape(H, W))
print('upper bounds:\n', ub.reshape(H, W))

l, u = SIM_csr.getRanges()
print('State bounds computed with getRanges() via LP solver:')
print('lower bounds:\n', l.reshape(H, W))
print('upper bounds:\n', u.reshape(H, W))

l, u = SIM_csr.estimateRanges()
print('State bounds computed with estimateRanges():')
print('lower bounds:\n', l.reshape(H, W))
print('upper bounds:\n', u.reshape(H, W))

l, u = SIM_csr.getRanges('estimate')
print('State bounds computed with getRanges(\'estimate\')')
print('lower bounds:\n', l.reshape(H, W))
print('upper bounds:\n', u.reshape(H, W))
```

```
 [10.  0.]]


Actual state bounds of SIM_csr:
lower bounds:
 [[0. 1.]
 [6. 0.]]
upper bounds:
 [[ 4.  5.]
 [10.  0.]]

State bounds computed with getRanges() via LP solver:
lower bounds:
 [[0. 1.]
 [6. 0.]]
upper bounds:
 [[ 4.  5.]
 [10.  0.]]

State bounds computed with estimateRanges():
lower bounds:
 [[0. 1.]
 [6. 0.]]
upper bounds:
 [[ 4.  5.]
 [10.  0.]]

State bounds computed with getRanges('estimate')
lower bounds:
 [[0. 1.]
 [6. 0.]]
upper bounds:
 [[ 4.  5.]
 [10.  0.]]
```

Chapter 4

Reachability Analysis

1 Fully Connected Layer

The fully connected operation is mathematically defined as $Y = WX + b$, where $X \in \mathbb{R}^n, W \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, Y \in \mathbb{R}^m$.

Lemma 1. *Given a fully connected layer $L_{fc} = \langle W, b \rangle$ with an input set $\mathcal{I} \in \{$Star, ProbStar$\}$, the reachability of a fully connected operation ($\texttt{fc}(\cdot)$) creates another reachable set $\mathcal{R} \in \{$Star, ProbStar$\}$ with the following characteristics:*

- ***For Star Sets**: If $\mathcal{I}$ is a Star set, then the output reachable set $\mathcal{R}$ is obtained by applying the affine mapping defined in Proposition 2.*

$$c' = Wc + b, \quad V' = \{Wv_1, Wv_2, \ldots, Wv_m\}, \quad P' \equiv P, \quad l' \equiv l, \quad u' \equiv u$$

- ***For ProbStar Sets**: If $\mathcal{I}$ is a ProbStar set, then the output reachable set $\mathcal{R}$ is obtained by applying the affine mapping defined in Proposition 7.*

$$c' = Wc + b, \quad V' = \{Wv_1, Wv_2, \ldots, Wv_m\}, \quad \mathcal{N}' = \mathcal{N}, \quad P' \equiv P, \quad l' \equiv l, \quad u' \equiv u$$

1.1 Fully connected layer construction

StarV supports multiple ways to create a fully connected layer, as shown in the code examples below.

Method 1: Construction using weight matrix and bias vector

In this method, the fully connected layer is constructed by explicitly specifying the weight matrix and bias vector. An example is provided below:

```
W = np.array([[1.0, -2.0],
              [-1., 0.5],
              [1., 1.5]])
b = np.array([0.5, 1.0, -0.5])
layer = [W, b]
L_fc = FullyConnectedLayer(layer)
L_fc.info()
```

```
# L_fc.info()
Layer type: FullyConnectedLayer
Weight matrix: [[ 1.  -2. ]
 [-1.   0.5]
 [ 1.   1.5]]
Bias vector: [ 0.5  1.  -0.5]
Input dimension: 2
Output dimension: 3
```

Method 2: Construction using random weight matrix and bias vector

This method generates a fully connected layer using a random weight matrix and bias vector:

```
L_fc = FullyConnectedLayer.rand(2, 3)
L_fc.info()
```

```
Layer type: FullyConnectedLayer
Weight matrix:
[[0.00429 0.69827]
 [0.13851 0.64468]
 [0.37437 0.18401]]
Bias vector: [0.45853 0.55195 0.86763]
Input dimension: 2
Output dimension: 3
```

Method 3: Construction using Pytorch layer

This method generates a fully connected layer using the PyTorch 'Linear' layer:

```
layer = torch.nn.Linear(2, 3)
L_fc = FullyConnectedLayer(layer)
L_fc.info()
```

```
Layer type: FullyConnectedLayer
Weight matrix:
[[-0.51537  0.03907]
 [-0.68917  0.17176]
 [-0.03965 -0.1751 ]]
Bias vector: [ 0.25125 -0.45769 -0.41249]
Input dimension: 2
Output dimension: 3
```

1.2 Fully connected layer reachability

This section demonstrates how to perform reachability analysis on a fully connected layer using various methods provided by StarV. Below are examples of evaluating an input vector and conducting reachability analysis with different configurations.

Evaluate an Input Vector on a Fully Connected Layer

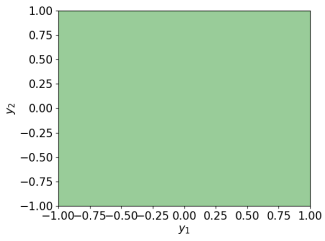
In this example, an input vector is evaluated through a fully connected layer. An example is provided below:

```
W = np.array([[1.0, -2.0],
              [-1., 0.5],
              [1., 1.5]])
b = np.array([0.5, 1.0, -0.5])
layer = [W, b]
L_fc = FullyConnectedLayer(layer)
L_fc.info()

x = np.array([1.0, 2.0])
y = L_fc.evaluate(x)

print('Input vector:', x)
print('Output vector:', y)
```

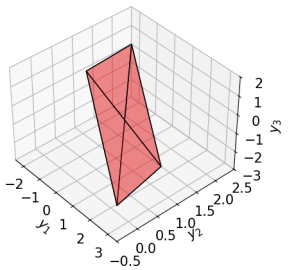
```
# L_fc.info()
Layer type: FullyConnectedLayer
Weight matrix:
[[ 1.  -2. ]
 [-1.   0.5]
 [ 1.   1.5]]
Bias vector: [ 0.5  1.  -0.5]
Input dimension: 2
Output dimension: 3

Input vector: [1. 2.]
Output vector: [-2.5  1.   3.5]
```

Reachability Analysis on Fully Connected Layer using Star. This approach performs reachability analysis using Star sets. An example is provided below:

```
W = np.array([[1.0, -2.0],
              [-1., 0.5],
              [1., 1.5]])
b = np.array([0.5, 1.0, -0.5])
layer = [W, b]
L_fc = FullyConnectedLayer(layer)

# Construct input set
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
S = Star(lb, ub)
I = [S]
print('\nInput (num of sets = {}):'.format(len(I)))
for I_i in I:
    print(I_i)
plot_star(I)

# Reachability analysis
R = L_fc.reach(I)
print('\nOutput (num of sets = {}):'.format(len(R)))
for R_i in R:
    print(R_i)
plot_3D_Star(R, qhull_option='QJ')
```



plot_star(inputs)



plot_3D_Star(outputs, qhull_option='QJ')

```
------------------------
Results
------------------------
Input (num of sets = 1):
```

```
Output (num of sets = 1):
Star Set:
V: [[ 0.5  1.  -2. ]
 [ 1.  -1.   0.5]
```

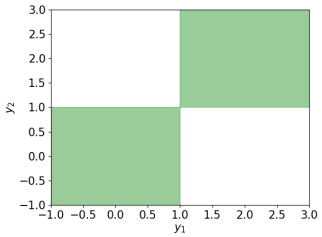
```
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
```

```
 [-0.5  1.   1.5]]
Predicate Constraints:
C: []
d: []
dim: 3
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
```

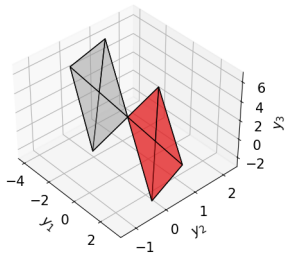
Reachability Analysis on Fully Connected Layer with Parallel Computing using Star

This method leverages parallel computing to enhance the efficiency of reachability analysis using Star sets. When handling multiple input sets, users have the option to perform the analysis without parallel computing if desired. An example is provided below:

```
W = np.array([[1.0, -2.0],
              [-1., 0.5],
              [1., 1.5]])
b = np.array([0.5, 1.0, -0.5])
layer = [W, b]
L_fc = FullyConnectedLayer(layer)

# Construct input set
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
S1 = Star(lb, ub)
S2 = Star(lb+2, ub+2)

I = [S1, S2]
print('\nInput (num of sets = {}):'.format(len(I)))
for I_i in I:
    print(I_i)
plot_star(I)

# Reachability analysis
numCores = 2
pool = multiprocessing.Pool(numCores)
R = L_fc.reach(I, pool=pool)
print('\nOutput (num of sets = {}):'.format(len(R)))
for R_i in R:
    print(R_i)
plot_3D_Star(R, qhull_option='QJ')
```



plot_star(inputs)



plot_3D_Star(outputs, qhull_option='QJ')

```
------------------------
Results
------------------------
Input (num of sets = 2):
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]

Star Set:
V: [[0. 2. 0.]
 [0. 0. 2.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
```

```
Output (num of sets = 2):
Star Set:
V: [[ 0.5  1.  -2. ]
 [ 1.  -1.   0.5]
 [-0.5  1.   1.5]]
Predicate Constraints:
C: []
d: []
dim: 3
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]

Star Set:
V: [[ 0.5  2.  -4. ]
 [ 1.  -2.   1. ]
 [-0.5  2.   3. ]]
Predicate Constraints:
C: []
d: []
dim: 3
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
```

Reachability Analysis on Fully Connected Layer using ProbStar

Reachability analysis is conducted using ProbStar sets in this example:

```
W = np.array([[1.0, -2.0],
              [-1., 0.5],
              [1., 1.5]])
b = np.array([0.5, 1.0, -0.5])
layer = [W, b]
L_fc = FullyConnectedLayer(layer)

# Construct input set
dim = 2
mu = np.zeros(dim)
Sig = np.eye(dim)
pred_lb = -np.ones(dim)
pred_ub = np.ones(dim)
P = ProbStar(mu, Sig, pred_lb, pred_ub)

I = [P]
print('\nInput (num of sets = {}):'.format(len(I)))
for I_i in I:
    print(I_i)

# Reachability analysis
R = L_fc.reach(I)
print('\nOutput (num of sets = {}):'.format(len(R)))
for R_i in R:
    print(R_i)
```

```
Input (num of sets = 1):
ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

Output (num of sets = 1):
ProbStar Set:
V: [[ 0.5  1.  -2. ]
 [ 1.  -1.   0.5]
 [-0.5  1.   1.5]]
Predicate Constraints:
C: []
d: []
dim: 3
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]
```

Reachability Analysis on Fully Connected Layer with Parallel Computing using ProbStar

This method combines ProbStar sets with parallel computing for efficient reachability analysis. When handling multiple input sets, users have the option to perform the analysis without parallel computing if desired. An example is provided below:

```
W = np.array([[1.0, -2.0],
              [-1., 0.5],
              [1., 1.5]])
b = np.array([0.5, 1.0, -0.5])
layer = [W, b]
L_fc = FullyConnectedLayer(layer)

# Construct input set
dim = 2
mu = np.zeros(dim)
Sig = np.eye(dim)
pred_lb = -np.ones(dim)
pred_ub = np.ones(dim)
P1 = ProbStar(mu, Sig, pred_lb, pred_ub)
P2 = ProbStar(mu, Sig * 0.5, pred_lb * 2, pred_ub * 2)

I = [P1, P2]
print('\nInput (num of sets = {}):'.format(len(I)))
for I_i in I:
    print(I_i)

# Reachability analysis
numCores = 2
pool = multiprocessing.Pool(numCores)
R = L_fc.reach(I, pool=pool)
print('\nOutput (num of sets = {}):'.format(len(R)))
for R_i in R:
    print(R_i)
```

```
------------------------
Results
------------------------
```

```
Output (num of sets = 2):
ProbStar Set:
V: [[ 0.5  1.  -2. ]
```

```
Input (num of sets = 2):
ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-2. -2.]
pred_ub: [2. 2.]
mu: [0. 0.]
Sig: [[0.5 0. ]
 [0.  0.5]]
```

```
 [ 1.  -1.   0.5]
 [-0.5  1.   1.5]]
Predicate Constraints:
C: []
d: []
dim: 3
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[ 0.5  1.  -2. ]
 [ 1.  -1.   0.5]
 [-0.5  1.   1.5]]
Predicate Constraints:
C: []
d: []
dim: 3
nVars: 2
pred_lb: [-2. -2.]
pred_ub: [2. 2.]
mu: [0. 0.]
Sig: [[0.5 0. ]
 [0.  0.5]]
```

2 Rectified Linear Unit (ReLU) Layer

This section outlines the various operations related to the ReLU layer L_r, including construction, evaluation, and reachability analysis using different methodologies. The ReLU activation function is defined by:

$$L_r(x) = ReLU(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

2.1 ReLU Layer Construction

An example of how to construct a ReLU layer is provided below:

```
# Construct ReLU layer
L_r = ReLULayer()
L_r.info()
```

```
# L_r.info()
Layer type: ReLULayer
```

2.2 ReLU Layer Reachability

Evaluate a ReLU Layer with a Given Input Vector

This example demonstrates how to evaluate a ReLU layer using a specific input vector. An example is provided below:

```
# Construct a ReLU layer
L_r = ReLULayer()
x = np.array([-1.0, 2.0, -3.0])
output = L_r.evaluate(x)

print("Input vector:", x)
print("Output after ReLU layer:", output)
```

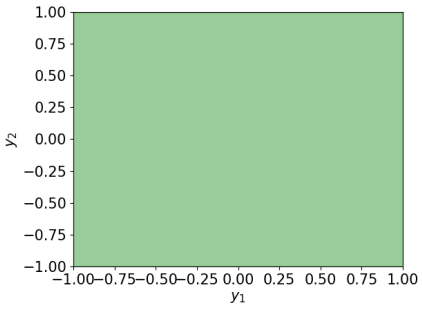
```
Input vector: [-1.  2. -3.]
Output after ReLU layer: [0. 2. 0.]
```

Perform exact reachability analysis on a ReLU layer using Star sets

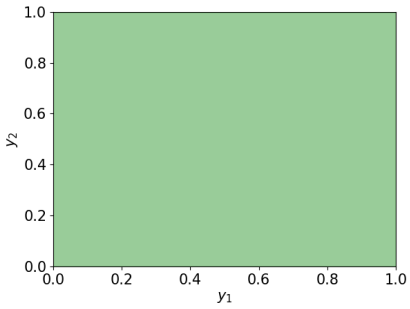
This approach conducts exact reachability analysis using Star sets. An example is provided below:

```
# Construct a ReLU layer
L_r = ReLULayer()

```

```
# Construct input set
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
S = Star(lb, ub)
inputs = [S]
print('\nInput sets (num of sets = {}):'.format(len(inputs)))
for input in inputs:
    print(input)
plot_star(inputs)

# Reachability analysis
outputs = L_r.reach(inputs, method='exact')
print('\nOutput sets (num of sets = {}):'.format(len(outputs)))
for output in outputs:
    print(output)
plot_star(outputs)
```



plot_star(inputs)



plot_star(outputs)

```
------------------------
Results
------------------------
Input sets (num of sets = 1):
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]

Output sets (num of sets = 4):
Star Set:
V: [[0. 0. 0.]
 [0. 0. 0.]]
Predicate Constraints:
C: [[0. 1.]
 [1. 0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [0. 0.]
```

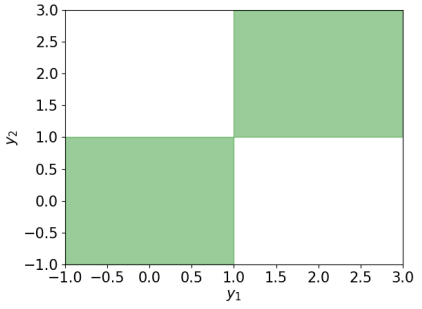
```
Star Set:
V: [[0. 0. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[ 0. -1.]
 [ 1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [-1.  0.]
pred_ub: [0. 1.]

Star Set:
V: [[0. 1. 0.]
 [0. 0. 0.]]
Predicate Constraints:
C: [[ 0.  1.]
 [-1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [ 0. -1.]
pred_ub: [1. 0.]
```

```
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[ 0. -1.]
 [-1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [0. 0.]
pred_ub: [1. 1.]
```

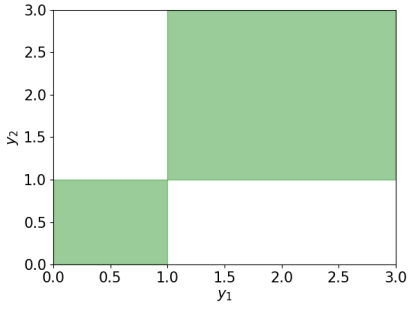
Perform Exact Reachability Analysis on a ReLU Layer with Parallel Computing Using Star Sets

This method leverages parallel computing to enhance the efficiency of exact reachability analysis with Star sets. When handling multiple input sets, users have the option to perform the analysis without parallel computing if desired. An example is provided below:

```
# Construct a ReLU layer
L_r = ReLULayer()

# Construct input set
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
S1 = Star(lb, ub)
S2 = Star(lb+2, ub+2)

inputs = [S1, S2]
print('\nInput sets (num of sets = {}):'.format(len(inputs)))
for input in inputs:
    print(input)

plot_star(inputs)

# Reachability analysis
pool = multiprocessing.Pool(2)
```



plot_star(inputs)



plot_star(outputs)

```
outputs = L_r.reach(inputs, method='exact', pool=pool)
print('\nOutput sets (num of sets = {}):'.format(len(outputs)))
for output in outputs:
    print(output)
plot_star(outputs)
```

```
------------------------
Results
------------------------
Input sets (num of sets = 2):
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]

Star Set:
V: [[2. 1. 0.]
 [2. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
```

```
Output sets (num of sets = 5):
Star Set:
V: [[0. 0. 0.]
 [0. 0. 0.]]
Predicate Constraints:
C: [[0. 1.]
 [1. 0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [0. 0.]

Star Set:
V: [[0. 0. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[ 0. -1.]
 [ 1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [-1.  0.]
pred_ub: [0. 1.]

Star Set:
V: [[0. 1. 0.]
 [0. 0. 0.]]
Predicate Constraints:
C: [[ 0.  1.]
 [-1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [ 0. -1.]
pred_ub: [1. 0.]
```

```
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[ 0. -1.]
 [-1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [0. 0.]
pred_ub: [1. 1.]

Star Set:
V: [[2. 1. 0.]
 [2. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
```

Perform Approximate Reachability Analysis on a ReLU Layer Using Star Sets

Lemma 2 (Over-approximation Rule for a ReLU function)**.** *For any input vector $x \in \mathbb{R}^n$, $l \leq x \leq u$ in which x_i is the i-th individual state of x, the output $y = L_r(x) = ReLU(x) \in \mathbb{R}^n$ satisfies the following rules:*

$$\begin{cases} y_i = x_i, & \text{if } l_i \geq 0, \\ y_i = 0, & \text{if } u_i \leq 0, \\ y_i \geq 0, y_i \leq \frac{u_i(x_i - l_i)}{u_i - l_i}, y_i \geq x_i, & \text{if } l_i < 0 \text{ and } u_i > 0. \end{cases}$$

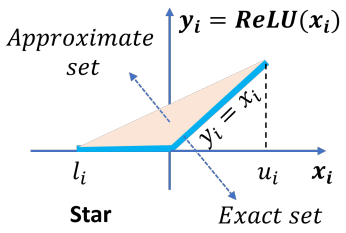
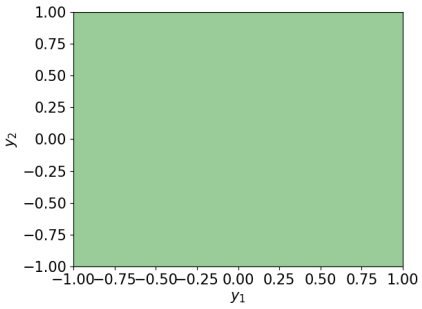


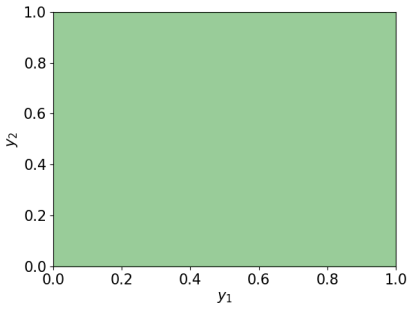
Figure 4.1: Exact and Over-approximation of ReLU.

Over-approximate reachability analysis for the ReLU function offers a balance between computational efficiency and precision, as illustrated in Fig. 4.1. An example is provided below. In this scenario, the user should perform reachability analysis directly on the created Star set without adding it to a list, resulting in a single Star set output.

```python
# Construct a ReLU layer
L_r = ReLULayer()

# Construct input set
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
input = Star(lb, ub)
print('\nInput set:')
print(input)
plot_star(input)

# Reachability analysis
output = L_r.reach(input, method='approx')
print('\nOutput set:')
print(output)
plot_star(output)
```



plot_star(input)



plot_star(output)

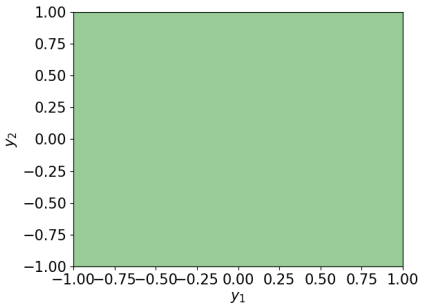
```
------------------------
Results
------------------------
Input set:
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
```

```
Output set:
Star Set:
V: [[0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
Predicate Constraints:
C: [[ 0.   0.  -1.  -0. ]
 [ 0.   0.  -0.  -1. ]
 [ 1.   0.  -1.  -0. ]
 [ 0.   1.  -0.  -1. ]
 [-0.5 -0.   1.   0. ]
 [-0.  -0.5  0.   1. ]]
d: [ 0.   0.  -0.  -0.   0.5  0.5]
dim: 2
nVars: 4
pred_lb: [-1. -1.  0.  0.]
pred_ub: [1. 1. 1. 1.]
```

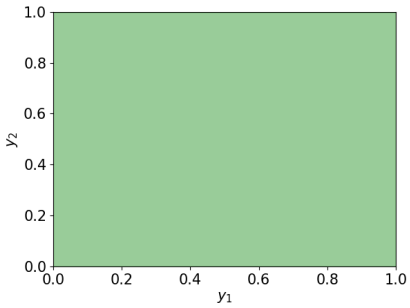
Perform Relaxed Reachability Analysis on a ReLU Layer Using Star Sets

Relaxed reachability analysis allows for broader approximations of the output set, facilitating faster computations. An example is provided below:

```python
# Construct a ReLU layer
L_r = ReLULayer()

# Construct input set
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
input = Star(lb, ub)
print('\nInput set:')
print(input)
plot_star(input)

# Reachability analysis
output = L_r.reach(input, method='approx',
↪  RF=0.4)
print('\nOutput set:')
print(output)
plot_star(output)
```



plot_star(input)



plot_star(output)

```
------------------------
Results
------------------------
Input set:
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
```

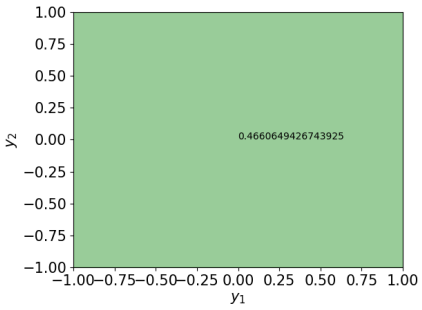
```
Output set:
Star Set:
V: [[0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
Predicate Constraints:
C: [[ 0.   0.  -1.  -0. ]
 [ 0.   0.  -0.  -1. ]
 [ 1.   0.  -1.  -0. ]
 [ 0.   1.  -0.  -1. ]
 [-0.5 -0.   1.   0. ]
 [-0.  -0.5  0.   1. ]]
d: [ 0.   0.  -0.  -0.   0.5  0.5]
dim: 2
nVars: 4
pred_lb: [-1. -1.  0.  0.]
```

```
pred_ub: [1. 1. 1. 1.]
```

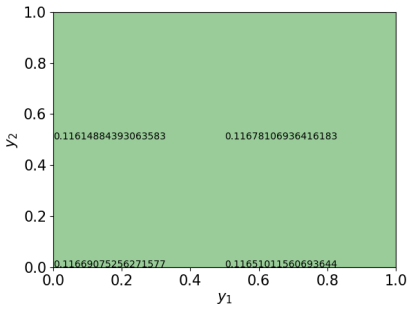
Perform Exact Reachability Analysis on a ReLU Layer Using ProbStar Sets

This method conducts exact reachability analysis with probabilistic star (ProbStar) sets. An example is provided below:

```
# Construct a ReLU layer
L_r = ReLULayer()

# Construct input set
dim = 2
mu = np.zeros(dim)
Sig = np.eye(dim)
pred_lb = -np.ones(dim)
pred_ub = np.ones(dim)
P = ProbStar(mu, Sig, pred_lb, pred_ub)
inputs = [P]
print('\nInput sets (num of sets = {}):'.format(len(inputs)))
for input in inputs:
    print(input)
plot_probstar(inputs)

# Reachability analysis
outputs = L_r.reach(inputs, method='exact')
print('\nOutput sets (num of sets = {}):'.format(len(outputs)))
for output in outputs:
    print(output)
plot_probstar(outputs)
```



plot_star(inputs)



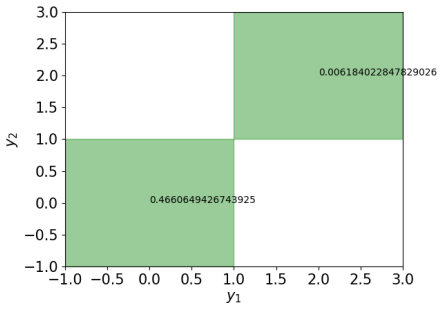
plot_star(outputs)

```
------------------------
Results
------------------------
Input sets (num of sets = 1):
ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

Output sets (num of sets = 4):
ProbStar Set:
V: [[0. 0. 0.]
 [0. 0. 0.]]
Predicate Constraints:
C: [[0. 1.]
 [1. 0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [0. 0.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[0. 0. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[ 0. -1.]
 [ 1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [-1.  0.]
pred_ub: [0. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 0.]]
Predicate Constraints:
C: [[ 0.  1.]
 [-1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [ 0. -1.]
pred_ub: [1. 0.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[ 0. -1.]
 [-1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [0. 0.]
pred_ub: [1. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]
```

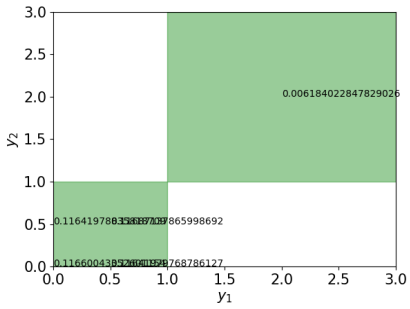
Perform Exact Reachability Analysis on a ReLU Layer with Parallel Computing Using ProbStar Sets

Combining ProbStar sets with parallel computing to optimize the performance of exact reachability analysis. The user can choose to run without parallel computing in the case of multiple input sets. An example is provided below:

```python
# Construct a ReLU layer
L_r = ReLULayer()

# Construct input set
dim = 2
mu = np.zeros(dim)
Sig = np.eye(dim)
pred_lb = -np.ones(dim)
pred_ub = np.ones(dim)
P1 = ProbStar(mu, Sig, pred_lb, pred_ub)
P2 = ProbStar(mu, Sig * 0.5, pred_lb + 2, pred_ub + 2)
inputs = [P1, P2]
print('\nInput sets (num of sets = {}):'.format(len(inputs)))
for input in inputs:
    print(input)
plot_probstar(inputs)

# Reachability analysis
pool = multiprocessing.Pool(2)
outputs = L_r.reach(inputs, method='exact', pool=pool)
print('\nOutput sets (num of sets = {}):'.format(len(outputs)))
for output in outputs:
    print(output)
plot_probstar(outputs)
```



plot_star(inputs)



plot_star(outputs)

```
------------------------
Results
------------------------
Input sets (num of sets = 2):
ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [1. 1.]
pred_ub: [3. 3.]
mu: [0. 0.]
Sig: [[0.5 0. ]
 [0.  0.5]]
```

```
Output sets (num of sets = 5):
ProbStar Set:
V: [[0. 0. 0.]
 [0. 0. 0.]]
Predicate Constraints:
C: [[0. 1.]
 [1. 0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [0. 0.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[0. 0. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[ 0. -1.]
 [ 1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [-1.  0.]
pred_ub: [0. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 0.]]
Predicate Constraints:
C: [[ 0.  1.]
 [-1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [ 0. -1.]
pred_ub: [1. 0.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]
```

```
ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[ 0. -1.]
 [-1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [0. 0.]
pred_ub: [1. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [1. 1.]
pred_ub: [3. 3.]
mu: [0. 0.]
Sig: [[0.5 0. ]
 [0.  0.5]]
```

3 Leaky Rectified Linear Unit (LeakyReLU) Layer

This section outlines the various operations related to the LeakyReLU layer L_{lr}, including construction, evaluation, and reachability analysis using different methodologies. The LeakyReLU activation function is defined by:

$$L_{lr}(\gamma, x) = LeakyReLU(\gamma, x) = \begin{cases} \gamma x & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

where γ is a small constant (e.g., $\gamma = 0.1$) that allows a small, non-zero gradient when the unit is inactive.

3.1 LeakyReLU Layer Construction

An example of how to construct a LeakyReLU layer is provided below:

```
# Construct LeakyReLU layer
L_lr = LeakyReLULayer()
L_lr.info()
```

```
# L_lr.info()
Layer type: LeakyReLULayer
```

3.2 LeakyReLU Layer Reachability

Evaluate a LeakyReLU Layer with a Given Input Vector

This example demonstrates how to evaluate a LeakyReLU layer using a specific input vector. An example is provided below:

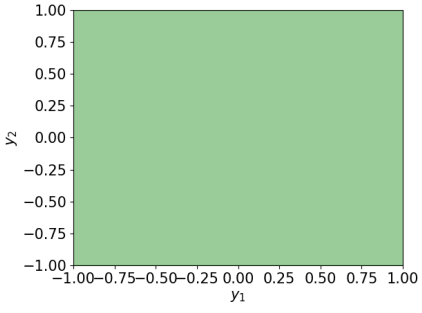
```
# Construct a LeakyReLU layer
L_lr = LeakyReLULayer()
x = np.array([-1.0, 2.0, -3.0])
output = L_lr.evaluate(x, gamma=0.1)

print("Input vector:", x)
print("Output after LeakyReLU layer:", output)
```

```
Input vector: [-1.  2. -3.]
Output after LeakyReLU layer: [-0.1  2.  -0.3]
```

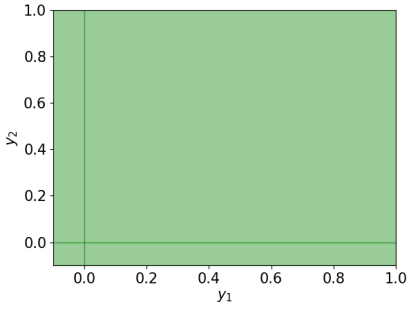
Perform exact reachability analysis on a LeakyReLU layer using Star sets

This approach conducts exact reachability analysis using Star sets. Users can choose to process multiple Star sets and optionally utilize parallel computing as needed, as illustrated in the ReLU layer reachability example.

```
# Construct a LeakyReLU layer
L_lr = LeakyReLULayer()

# Construct input set
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
S = Star(lb, ub)
inputs = [S]
print('\nInput sets (num of sets = {}):'.format(len(inputs)))
for input in inputs:
    print(input)
plot_star(inputs)

# Reachability analysis
outputs = L_lr.reach(inputs, method='exact')
print('\nOutput sets (num of sets = {}):'.format(len(outputs)))
for output in outputs:
    print(output)
plot_star(outputs)
```



plot_star(inputs)



plot_star(outputs)

```
------------------------
Results
------------------------
Input sets (num of sets = 1):
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]

Output sets (num of sets = 4):
Star Set:
V: [[0.  0.1 0. ]
 [0.  0.  0.1]]
Predicate Constraints:
C: [[0. 1.]
 [1. 0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [0. 0.]
```

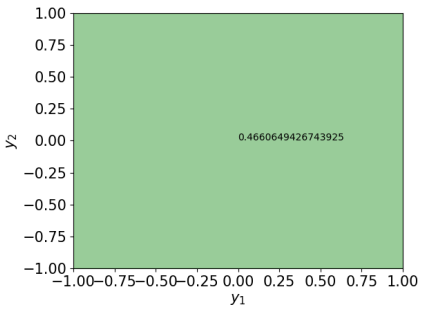
```
Star Set:
V: [[0.  0.1 0. ]
 [0.  0.  1. ]]
Predicate Constraints:
C: [[ 0. -1.]
 [ 1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [-1.  0.]
pred_ub: [0. 1.]

Star Set:
V: [[0.  1.  0. ]
 [0.  0.  0.1]]
Predicate Constraints:
C: [[ 0.  1.]
 [-1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [ 0. -1.]
pred_ub: [1. 0.]
```

```
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[ 0. -1.]
 [-1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [0. 0.]
pred_ub: [1. 1.]
```

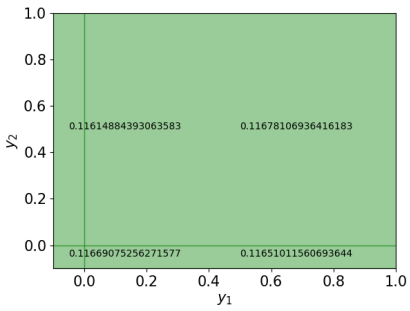
Perform Exact Reachability Analysis on a LeakyReLU Layer Using ProbStar Sets

This method conducts exact reachability analysis with probabilistic star (ProbStar) sets. Users can choose to process multiple ProbStar sets and optionally utilize parallel computing as needed, as illustrated in the ReLU layer reachability example.

```
# Construct a LeakyReLU layer
L_lr = LeakyReLULayer()

# Construct input set
dim = 2
mu = np.zeros(dim)
Sig = np.eye(dim)
pred_lb = -np.ones(dim)
pred_ub = np.ones(dim)
P = ProbStar(mu, Sig, pred_lb, pred_ub)
inputs = [P]
print('\nInput sets (num of sets = {}):'.format(len(inputs)))
for input in inputs:
    print(input)
plot_probstar(inputs)

# Reachability analysis
outputs = L_lr.reach(inputs, method='exact')
print('\nOutput sets (num of sets = {}):'.format(len(outputs)))
for output in outputs:
    print(output)
plot_probstar(outputs)
```



plot_probstar(inputs)



plot_probstar(outputs)

```
------------------------
Results
------------------------
Input sets (num of sets = 1):
ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
```

```
ProbStar Set:
V: [[0.  0.1 0. ]
 [0.  0.  1. ]]
Predicate Constraints:
C: [[ 0. -1.]
 [ 1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [-1.  0.]
pred_ub: [0. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
```

```
ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[ 0. -1.]
 [-1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [0. 0.]
pred_ub: [1. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]
```

```
 [0. 1.]]

Output sets (num of sets = 4):
ProbStar Set:
V: [[0.  0.1 0. ]
 [0.  0.  0.1]]
Predicate Constraints:
C: [[0. 1.]
 [1. 0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [0. 0.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]
V: [[0.  1.  0. ]
 [0.  0.  0.1]]
Predicate Constraints:
C: [[ 0.  1.]
 [-1.  0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [ 0. -1.]
pred_ub: [1. 0.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]
```

4 Saturated Linear (SatLin) Layer

This section outlines the various operations related to the SatLIn layer L_{st}, including construction, evaluation, and reachability analysis using different methodologies. The SatLin activation function is defined by:

$$L_{st}(x) = SatLin(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } 0 \leq x \leq 1 \\ 1 & \text{if } x > 1 \end{cases}$$

4.1 SatLin Layer Construction

An example of how to construct a SatLin layer is provided below:

```
# Construct SatLin layer
L_st = SatLinLayer()
L_st.info()
```

```
# L_st.info()
Layer type: SatLinLayer
```

4.2 SatLin Layer Reachability

Evaluate a SatLin Layer with a Given Input Vector

This example demonstrates how to evaluate a SatLin layer using a specific input vector. An example is provided below:

```
# Construct a SatLin layer
L_st = SatLinLayer()
x = np.array([-1.0, 2.0, -3.0, 0.5])
output = L_st.evaluate(x)

print("Input vector:", x)
print("Output after SatLin layer:", output)
```

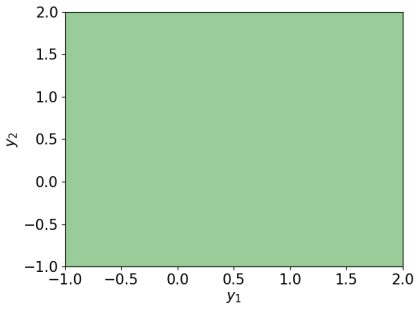
```
Input vector: [-1.   2.  -3.   0.5]
Output after SatLin layer: [0.  1.  0.  0.5]
```

Perform exact reachability analysis on a SatLin layer using Star sets.

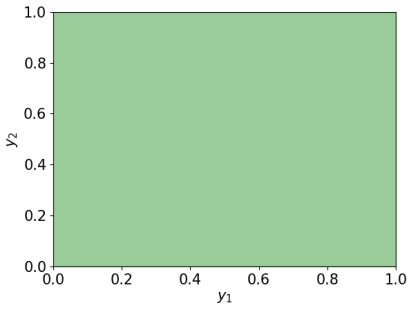
This approach conducts exact reachability analysis using Star sets. Users can choose to process multiple Star sets and optionally utilize parallel computing as needed, as illustrated in the ReLU layer reachability example.

```
# Construct a SatLin layer
L_st = SatLinLayer()

# Construct input set
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
S = Star(lb, ub+1)
inputs = [S]

```

```
print('\nInput sets (num of sets = {}):'.format(len(inputs)))
for input in inputs:
    print(input)
plot_star(inputs)

# Reachability analysis
outputs = L_st.reach(inputs, method='exact')

print('\nOutput sets (num of sets = {}):'.format(len(outputs)))
for output in outputs:
    print(output)
plot_star(outputs)
```



plot_star(inputs)



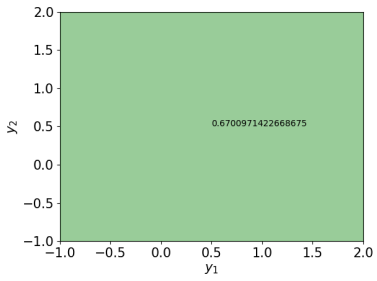
plot_star(outputs)

```
-------------------------
Results
-------------------------
Input sets (num of sets = 1):
Star Set:
V: [[0.5 1.5 0. ]
 [0.5 0.  1.5]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]

Output sets (num of sets = 9):
Star Set:
V: [[0. 0. 0.]
 [0. 0. 0.]]
Predicate Constraints:
C: [[0.  1.5]
 [1.5 0. ]]
d: [-0.5 -0.5]
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [-0.33333 -0.33333]

Star Set:
V: [[0.  0.  0. ]
 [0.5 0.  1.5]]
Predicate Constraints:
C: [[ 0.   1.5]
 [ 0.  -1.5]
 [ 1.5  0. ]]
d: [ 0.5  0.5 -0.5]
dim: 2
nVars: 2
pred_lb: [-1.      -0.33333]
pred_ub: [-0.33333  0.33333]

Star Set:
V: [[0. 0. 0.]
 [1. 0. 0.]]
Predicate Constraints:
C: [[ 0.  -1.5]
 [ 1.5  0. ]]
d: [-0.5 -0.5]
dim: 2
nVars: 2
pred_lb: [-1.       0.33333]
pred_ub: [-0.33333  1.     ]

Star Set:
V: [[0.5 1.5 0. ]
 [0.  0.  0. ]]
Predicate Constraints:
C: [[ 0.   1.5]
 [ 1.5  0. ]
 [-1.5  0. ]]
d: [-0.5  0.5  0.5]
dim: 2
nVars: 2
pred_lb: [-0.33333 -1.     ]
pred_ub: [ 0.33333 -0.33333]

Star Set:
V: [[0.5 1.5 0. ]
 [0.5 0.  1.5]]
Predicate Constraints:
C: [[ 0.   1.5]
 [ 0.  -1.5]
 [ 1.5  0. ]
 [-1.5  0. ]]
d: [0.5 0.5 0.5 0.5]
dim: 2
nVars: 2
pred_lb: [-0.33333 -0.33333]
pred_ub: [0.33333 0.33333]

Star Set:
V: [[0.5 1.5 0. ]
 [1.  0.  0. ]]
Predicate Constraints:
C: [[ 0.  -1.5]
 [ 1.5  0. ]
 [-1.5  0. ]]
d: [-0.5  0.5  0.5]
dim: 2
nVars: 2
pred_lb: [-0.33333  0.33333]
pred_ub: [0.33333 1.     ]

Star Set:
V: [[1. 0. 0.]
 [0. 0. 0.]]
Predicate Constraints:
C: [[ 0.   1.5]
 [-1.5  0. ]]
d: [-0.5 -0.5]
dim: 2
nVars: 2
pred_lb: [ 0.33333 -1.     ]
pred_ub: [ 1.      -0.33333]

Star Set:
V: [[1.  0.  0. ]
 [0.5 0.  1.5]]
Predicate Constraints:
C: [[ 0.   1.5]
 [ 0.  -1.5]
 [-1.5  0. ]]
d: [ 0.5  0.5 -0.5]
dim: 2
nVars: 2
pred_lb: [ 0.33333 -0.33333]
pred_ub: [1.      0.33333]

Star Set:
V: [[1. 0. 0.]
 [1. 0. 0.]]
Predicate Constraints:
C: [[ 0.  -1.5]
 [-1.5  0. ]]
d: [-0.5 -0.5]
dim: 2
nVars: 2
pred_lb: [0.33333 0.33333]
pred_ub: [1. 1.]
```

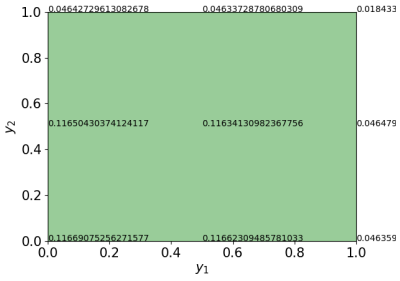
Perform Exact Reachability Analysis on a SatLin Layer Using ProbStar Sets

This method conducts exact reachability analysis with probabilistic star (ProbStar) sets. As illustrated in the ReLU layer reachability example, users can choose to process multiple ProbStar sets and optionally utilize parallel computing as needed.

```
# Construct a SatLin layer
L_st = SatLinLayer()

# Construct input set
dim = 2
mu = np.zeros(dim)
Sig = np.eye(dim)
pred_lb = -np.ones(dim)
pred_ub = np.ones(dim)
P = ProbStar(mu, Sig, pred_lb, pred_ub+1)
inputs = [P]

```



plot_star(inputs)



plot_star(outputs)

```
print('\nInput sets (num of sets = {}):'.format(len(inputs)))
for input in inputs:
    print(input)
plot_probstar(inputs)

# Reachability analysis
outputs = L_st.reach(inputs, method='exact')
print('\nOutput sets (num of sets = {}):'.format(len(outputs)))
for output in outputs:
    print(output)
plot_probstar(outputs)
```

```
-------------------------
Results
-------------------------
Input sets (num of sets = 1):
ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [2. 2.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

Output sets (num of sets = 9):
ProbStar Set:
V: [[0. 0. 0.]
 [0. 0. 0.]]
Predicate Constraints:
C: [[0. 1.]
 [1. 0.]]
d: [0. 0.]
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [0. 0.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[0. 0. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[ 0.  1.]
 [ 0. -1.]
 [ 1.  0.]]
d: [1. 0. 0.]
dim: 2
nVars: 2
pred_lb: [-1.  0.]
pred_ub: [0. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[0. 0. 0.]
 [1. 0. 0.]]
Predicate Constraints:
C: [[ 0. -1.]
 [ 1.  0.]]
d: [-1.  0.]
dim: 2
nVars: 2
pred_lb: [-1.  1.]
pred_ub: [0. 2.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 0.]]
Predicate Constraints:
C: [[ 0.  1.]
 [ 1.  0.]
 [-1.  0.]]
d: [0. 1. 0.]
dim: 2
nVars: 2
pred_lb: [ 0. -1.]
pred_ub: [1. 0.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[ 0.  1.]
 [ 0. -1.]
 [ 1.  0.]
 [-1.  0.]]
d: [1. 0. 1. 0.]
dim: 2
nVars: 2
pred_lb: [0. 0.]
pred_ub: [1. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[0. 1. 0.]
 [1. 0. 0.]]
Predicate Constraints:
C: [[ 0. -1.]
 [ 1.  0.]
 [-1.  0.]]
d: [-1.  1.  0.]
dim: 2
nVars: 2
pred_lb: [0. 1.]
pred_ub: [1. 2.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[1. 0. 0.]
 [0. 0. 0.]]
Predicate Constraints:
C: [[ 0.  1.]
 [-1.  0.]]
d: [ 0. -1.]
dim: 2
nVars: 2
pred_lb: [ 1. -1.]
pred_ub: [2. 0.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[1. 0. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[ 0.  1.]
 [ 0. -1.]
 [-1.  0.]]
d: [ 1.  0. -1.]
dim: 2
nVars: 2
pred_lb: [1. 0.]
pred_ub: [2. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[1. 0. 0.]
 [1. 0. 0.]]
Predicate Constraints:
C: [[ 0. -1.]
 [-1.  0.]]
d: [-1. -1.]
dim: 2
nVars: 2
pred_lb: [1. 1.]
pred_ub: [2. 2.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]
```

5 Symmetric Saturated Linear (Satlins) Layer

This section outlines the various operations related to the SatLins layer L_{sts}, including construction, evaluation, and reachability analysis using different methodologies. The SatLins activation function is defined by:

$$L_{sts}(x) = SatLins(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 \leq x \leq 1 \\ 1 & \text{if } x > 1 \end{cases}$$

5.1 SatLins Layer Construction

An example of how to construct a SatLins layer is provided below:

```
# Construct SatLins layer
L_sts = SatLinsLayer()
L_sts.info()
```

```
# L_sts.info()
Layer type: SatLinsLayer
```

5.2 SatLins Layer Reachability

Evaluate a SatLins Layer with a Given Input Vector

This example demonstrates how to evaluate a SatLins layer using a specific input vector. An example is provided below:

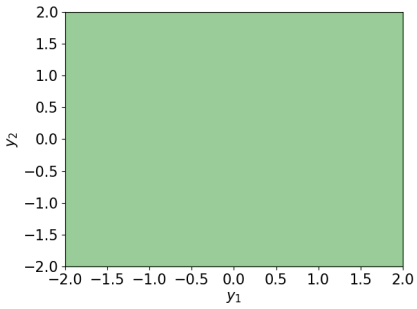
```
# Construct a SatLins layer
L_sts = SatLinsLayer()
x = np.array([-1.5, 2.0, -0.5, 0.5])
output = L_sts.evaluate(x)

print("Input vector:", x)
print("Output after SatLins layer:", output)
```

```
Input vector: [-1.5  2.  -0.5  0.5]
Output after SatLins layer: [-1.   1.  -0.5  0.5]
```

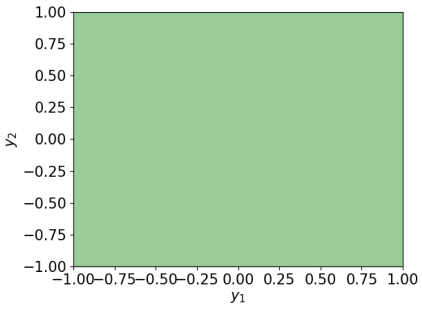
Perform exact reachability analysis on a SatLins layer using Star sets

This approach conducts exact reachability analysis using Star sets. Users can choose to process multiple Star sets and optionally utilize parallel computing as needed, as illustrated in the ReLU layer reachability example.

```
# Construct a SatLins layer
L_sts = SatLinsLayer()

# Construct input set
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
S = Star(lb-1, ub+1)
inputs = [S]
print('\nInput sets (num of sets = {}):'.format(len(inputs)))
for input in inputs:
    print(input)
plot_star(inputs)

# Reachability analysis
outputs = L_sts.reach(inputs, method='exact')
print('\nOutput sets (num of sets = {}):'.format(len(outputs)))
for output in outputs:
    print(output)
plot_star(outputs)
```



plot_star(inputs)



plot_star(outputs)

```
-------------------------
Results
-------------------------
Input sets (num of sets = 1):
Star Set:
V: [[0. 2. 0.]
 [0. 0. 2.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]

Output sets (num of sets = 9):
Star Set:
V: [[-1.  0.  0.]
```

```
Star Set:
V: [[-1.  0.  0.]
 [ 1.  0.  0.]]
Predicate Constraints:
C: [[ 0. -2.]
 [ 2.  0.]]
d: [-1. -1.]
dim: 2
nVars: 2
pred_lb: [-1.   0.5]
pred_ub: [-0.5  1. ]

Star Set:
V: [[ 0.  2.  0.]
 [-1.  0.  0.]]
Predicate Constraints:
C: [[ 0.  2.]
 [ 2.  0.]]
```

```
Star Set:
V: [[0. 2. 0.]
 [1. 0. 0.]]
Predicate Constraints:
C: [[ 0. -2.]
 [ 2.  0.]
 [-2.  0.]]
d: [-1.  1.  1.]
dim: 2
nVars: 2
pred_lb: [-0.5  0.5]
pred_ub: [0.5 1. ]

Star Set:
V: [[ 1.  0.  0.]
 [-1.  0.  0.]]
Predicate Constraints:
C: [[ 0.  2.]
```

```
Star Set:
V: [[1. 0. 0.]
 [1. 0. 0.]]
Predicate Constraints:
C: [[ 0. -2.]
 [-2.  0.]]
d: [-1. -1.]
dim: 2
nVars: 2
pred_lb: [0.5 0.5]
pred_ub: [1. 1.]
```

```
 [-1.  0.  0.]]
Predicate Constraints:
C: [[0. 2.]
 [2. 0.]]
d: [-1. -1.]
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [-0.5 -0.5]

Star Set:
V: [[-1.  0.  0.]
 [ 0.  0.  2.]]
Predicate Constraints:
C: [[ 0.  2.]
 [ 0. -2.]
 [ 2.  0.]]
d: [ 1.  1. -1.]
dim: 2
nVars: 2
pred_lb: [-1.  -0.5]
pred_ub: [-0.5  0.5]
```

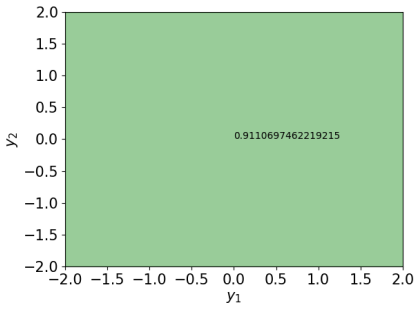
```
 [-2.  0.]]
d: [-1.  1.  1.]
dim: 2
nVars: 2
pred_lb: [-0.5 -1. ]
pred_ub: [ 0.5 -0.5]

Star Set:
V: [[0. 2. 0.]
 [0. 0. 2.]]
Predicate Constraints:
C: [[ 0.  2.]
 [ 0. -2.]
 [ 2.  0.]
 [-2.  0.]]
d: [1. 1. 1. 1.]
dim: 2
nVars: 2
pred_lb: [-0.5 -0.5]
pred_ub: [0.5 0.5]
```

```
 [-2.  0.]]
d: [-1. -1.]
dim: 2
nVars: 2
pred_lb: [ 0.5 -1. ]
pred_ub: [ 1.  -0.5]

Star Set:
V: [[1. 0. 0.]
 [0. 0. 2.]]
Predicate Constraints:
C: [[ 0.  2.]
 [ 0. -2.]
 [-2.  0.]]
d: [ 1.  1. -1.]
dim: 2
nVars: 2
pred_lb: [ 0.5 -0.5]
pred_ub: [1.  0.5]
```

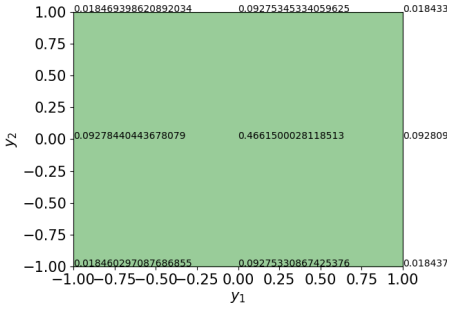
Perform Exact Reachability Analysis on a SatLins Layer Using ProbStar Sets

This method conducts exact reachability analysis with probabilistic star (ProbStar) sets. As illustrated in the ReLU layer reachability example, users can choose to process multiple ProbStar sets and optionally utilize parallel computing as needed.

```
# Construct a SatLins layer
L_sts = SatLinsLayer()

# Construct input set
dim = 2
mu = np.zeros(dim)
Sig = np.eye(dim)
pred_lb = -np.ones(dim)
pred_ub = np.ones(dim)
P = ProbStar(mu, Sig, pred_lb, pred_ub+1)
inputs = [P]

print('\nInput sets (num of sets = {}):'.format(len(inputs)))
for input in inputs:
    print(input)
plot_probstar(inputs)

# Reachability analysis
outputs = L_sts.reach(inputs, method='exact')
print('\nOutput sets (num of sets = {}):'.format(len(outputs)))
for output in outputs:
    print(output)
plot_probstar(outputs)
```



plot_star(inputs)



plot_star(outputs)

```
------------------------
Results
------------------------
Input sets (num of sets = 1):
ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-2. -2.]
pred_ub: [2. 2.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

Output sets (num of sets = 9):
ProbStar Set:
```

```
ProbStar Set:
V: [[-1.  0.  0.]
 [ 1.  0.  0.]]
Predicate Constraints:
C: [[ 0. -1.]
 [ 1.  0.]]
d: [-1. -1.]
dim: 2
nVars: 2
pred_lb: [-2.  1.]
pred_ub: [-1.  2.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[ 0.  1.  0.]
 [-1.  0.  0.]]
Predicate Constraints:
C: [[ 0.  1.]
```

```
ProbStar Set:
V: [[0. 1. 0.]
 [1. 0. 0.]]
Predicate Constraints:
C: [[ 0. -1.]
 [ 1.  0.]
 [-1.  0.]]
d: [-1.  1.  1.]
dim: 2
nVars: 2
pred_lb: [-1.  1.]
pred_ub: [1. 2.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[ 1.  0.  0.]
 [-1.  0.  0.]]
Predicate Constraints:
```

```
ProbStar Set:
V: [[1. 0. 0.]
 [1. 0. 0.]]
Predicate Constraints:
C: [[ 0. -1.]
 [-1.  0.]]
d: [-1. -1.]
dim: 2
nVars: 2
pred_lb: [1. 1.]
pred_ub: [2. 2.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]
```

```
V: [[-1.  0.  0.]
 [-1.  0.  0.]]
Predicate Constraints:
C: [[0. 1.]
 [1. 0.]]
d: [-1. -1.]
dim: 2
nVars: 2
pred_lb: [-2. -2.]
pred_ub: [-1. -1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[-1.  0.  0.]
 [ 0.  0.  1.]]
Predicate Constraints:
C: [[ 0.  1.]
 [ 0. -1.]
 [ 1.  0.]]
d: [ 1.  1. -1.]
dim: 2
nVars: 2
pred_lb: [-2. -1.]
pred_ub: [-1.  1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]
```

```
 [ 1.  0.]
 [-1.  0.]]
d: [-1.  1.  1.]
dim: 2
nVars: 2
pred_lb: [-1. -2.]
pred_ub: [ 1. -1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[ 0.  1.]
 [ 0. -1.]
 [ 1.  0.]
 [-1.  0.]]
d: [1. 1. 1. 1.]
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]
```

```
C: [[ 0.  1.]
 [-1.  0.]]
d: [-1. -1.]
dim: 2
nVars: 2
pred_lb: [ 1. -2.]
pred_ub: [ 2. -1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]

ProbStar Set:
V: [[1. 0. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: [[ 0.  1.]
 [ 0. -1.]
 [-1.  0.]]
d: [ 1.  1. -1.]
dim: 2
nVars: 2
pred_lb: [ 1. -1.]
pred_ub: [2. 1.]
mu: [0. 0.]
Sig: [[1. 0.]
 [0. 1.]]
```

6 Logistic Sigmoid (Sigmoid) Layer

Both Sigmoid (L_σ) and TanH (L_ϕ) layers follow the same sigmoidal over-approximation rule, as shown in the Lemma 3 below.

Lemma 3 (Star Overapproximation Rule for a Sigmoidal Function)**.** *For any input vector $x \in \mathbb{R}^n$, $l \leq x \leq u$ in which x_i is i-th individual state of x, the output $y = f^\sigma(x) \in \mathbb{R}^n$ satisfies the following rules:*
if $l_i == u_i$

$$y_i = f^\sigma(x_i) \tag{a}$$

if $l_i \geq 0$

$$\begin{cases} y_i \geq y_1 = \frac{f^\sigma(u_i) - f^\sigma(l_i)}{u_i - l_i} \times (x_i - l_i) + f^\sigma(l_i), \\ y_i \leq y_2 = f^{\sigma'}(u_i) \times (x_i - u_i) + f^\sigma(u_i), \\ y_i \leq y_3 = f^{\sigma'}(l_i) \times (x_i - l_i) + f^\sigma(l_i). \end{cases} \tag{b}$$

if $u_i \leq 0$

$$\begin{cases} y_i \geq y_1 = f^{\sigma'}(l_i) \times (x_i - l_i) + f^\sigma(l_i), \\ y_i \leq y_2 = \frac{f^\sigma(u_i) - f^\sigma(l_i)}{u_i - l_i} \times (x_i - l_i) + f^\sigma(l_i), \\ y_i \geq y_3 = f^{\sigma'}(u_i) \times (x_i - u_i) + f^\sigma(u_i). \end{cases} \tag{c}$$

if $l_i < 0$ and $u_i > 0$

$$\begin{cases} y_i \geq y_1 = \lambda \times (x_i - l_i) + f^\sigma(l_i), \\ y_i \leq y_2 = \lambda \times (x_i - u_i) + f^\sigma(u_i), \\ y_i \geq y_3 = \mu_l \times x_i - \mu_l \times u_i + f^\sigma(u_i), \\ y_i \leq y_4 = \mu_u \times x_i - \mu_u \times l_i + f^\sigma(l_i). \end{cases} \tag{d}$$

where $\lambda = min(f^{\sigma'}(l_i), f^{\sigma'}(u_i)), \mu_u = \frac{f^\sigma(l_i) - a_y}{l_i - a_x}, \mu_l = \frac{f^\sigma(u_i) - b_y}{u_i - b_x}, a_y = f^{\sigma'}(0) \times a_x + f^\sigma(0), a_x = \frac{f^\sigma(u_i) - \lambda \times u_i - f^\sigma(0)}{f^{\sigma'}(0) - \lambda}, b_y = f^{\sigma'}(0) \times b_x + f^\sigma(0), b_x = \frac{f^\sigma(l_i) - \lambda \times l_i - f^\sigma(0)}{f^{\sigma'}(0) - \lambda}$

6.1 Logistic Sigmoid (Sigmoid) Layer Construction

```
# Construct LogSig (Sigmoid) layer
L_sigmoid = LogSigLayer()
print(L_sigmoid)
```

```
#print(L_sigmoid)
Layer type: LogSigLayer
```

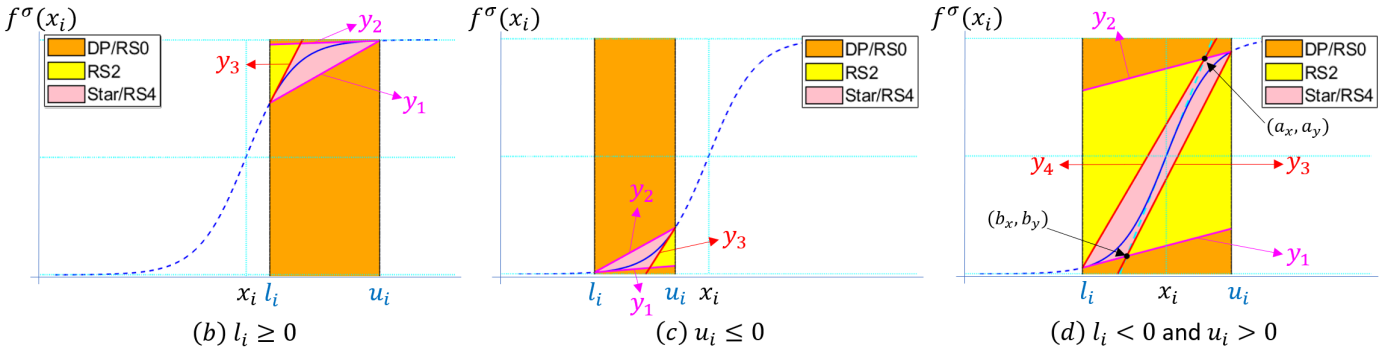


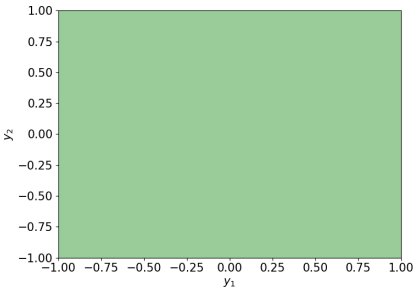
Figure 4.2: **Over-approximation of TanH/Sigmoid**.

6.2 Logistic Sigmoid (Sigmoid) Layer Approximate Reachability

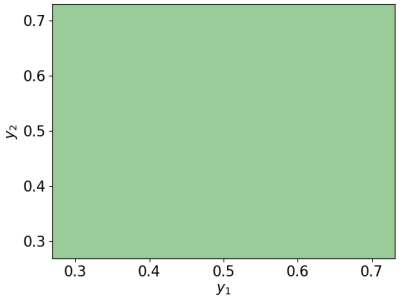
Logistic Sigmoid (Sigmoid) layer over-approximate reachability analysis with **Star**.

```
# Construct a Sigmoid layer
L_sigmoid = LogSigLayer()

# Construct input set
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
input = Star(lb, ub)
print('\nInput set:')
print(input)
plot_star(input)

# Reachability analysis
output = L_sigmoid.reach(input, method='approx')
print('\nOutput set:')
print(output)
plot_star(output)
```

```
Input set:
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]

Output set:
Star Set:
V: [[0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
Predicate Constraints:
C: [[ 0.19661  0.      -1.      -0.     ]
 [ 0.       0.19661 -0.      -1.     ]
 [-0.19661 -0.       1.       0.     ]
 [-0.      -0.19661  0.       1.     ]
 [ 0.36044  0.      -1.      -0.     ]
 [ 0.       0.36044 -0.      -1.     ]
 [-0.23798 -0.       1.       0.     ]
 [-0.      -0.23798  0.       1.     ]]
d: [-0.46555 -0.46555  0.53445  0.53445
    -0.37061 -0.37061  0.50692  0.50692]
dim: 2
nVars: 4
pred_lb: [-1.   -1.   0.26894   0.26894]
pred_ub: [ 1.    1.   0.73106   0.73106]
```



plot_star(input)

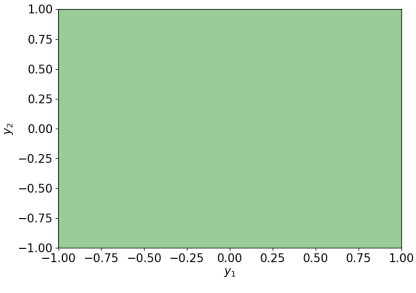


plot_star(output)

Logistic Sigmoid (Sigmoid) layer over-approximate reachability analysis with **SparseStar**.

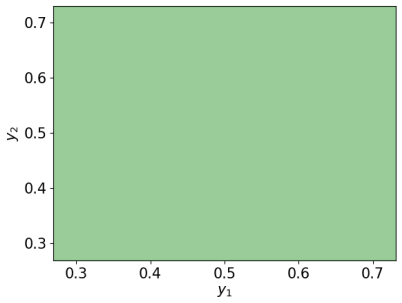
```
# Construct a TanH layer
L_sigmoid = LogSigLayer()

# Construct input set
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
input = SparseStar(lb, ub)
print('\nInput set:')
print(input)
plot_star(input)

# Reachability analysis
output = L_sigmoid.reach(input, method='approx')
print('\nOutput set:')
print(output)
plot_star(output)
```

```
Input set:
SparseStar Set:
A:
[[0. 1. 0.]
 [0. 0. 1.]]
C_csc:
[]
d: []
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
pred_depth: [0. 0.]
dim: 2
nVars: 2
nZVars: 0
nIVars: 2

Output set:
SparseStar Set:
A:
[[0. 1. 0.]
 [0. 0. 1.]]
C_csc:
[[ 0.19661  0.      -1.       0.     ]
```



plot_star(input)

```
 [ 0.       0.19661  0.       -1.     ]
 [-0.19661  0.       1.        0.     ]
 [ 0.      -0.19661  0.        1.     ]
 [ 0.23849  0.      -1.        0.     ]
 [ 0.       0.23849  0.       -1.     ]
 [-0.23849  0.       1.        0.     ]
 [ 0.      -0.23849  0.        1.     ]]
d: [-0.46555 -0.46555  0.53445  0.53445
↪  -0.49257 -0.49257  0.50743  0.50743]
pred_lb: [-1.   -1.   0.26894   0.26894]
pred_ub: [ 1.    1.   0.73106   0.73106]
pred_depth: [1. 1. 0. 0.]
dim: 2
nVars: 4
nZVars: 2
nIVars: 2
```



plot_star(output)

7 Hyperbolic Tangent (TanH) Layer

7.1 Hyperbolicx Tanget (TanH) Layer Construction

```
# Construct a Hyperbolic Tangent (TanH) layer
L_tanh = TanSigLayer()
print(L_tanh)
```

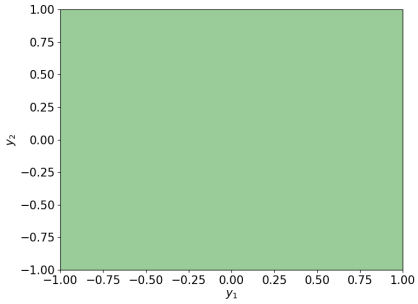
```
#print(L_tanh)
ayer type: TanSigLayer
```

7.2 Hyperbolic Tangent (TanH) Layer Approximate Reachability

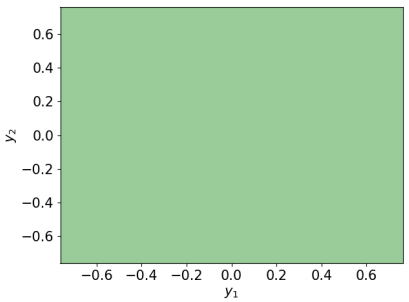
Hyperbolic Tangent (TanH) layer over-approximate reachability analysis with **Star**.

```
# Construct a TanH layer
L_tanh = TanSigLayer()

# Construct input set
lb = np.array([-1.0, -1.0])
ub = np.array([1.0, 1.0])
input = Star(lb, ub)
print('\nInput set:')
print(input)
plot_star(input)

# Reachability analysis
output = L_tanh.reach(input,
↪  method='approx')
print('\nOutput set:')
print(output)
plot_star(output)
```

```
Input set:
Star Set:
V: [[0. 1. 0.]
 [0. 0. 1.]]
Predicate Constraints:
C: []
d: []
dim: 2
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [ 1.  1.]

Output set:
Star Set:
V: [[0. 0. 0. 1. 0.]
    [0. 0. 0. 0. 1.]]
Predicate Constraints:
C: [[ 0.41997  0.      -1.      -0.     ]
 [ 0.       0.41997 -0.       -1.     ]
 [-0.41997 -0.        1.        0.     ]
 [-0.      -0.41997  0.        1.     ]
 [ 0.84996  0.      -1.       -0.     ]
 [ 0.       0.84996 -0.       -1.     ]
 [-0.84996 -0.        1.        0.     ]
 [-0.      -0.84996  0.        1.     ]]
d: [0.34162 0.34162 0.34162 0.34162 0.08837
↪  0.08837 0.08837 0.08837]
dim: 2
nVars: 4
pred_lb: [-1.   -1.   -0.76159   -0.76159]
pred_ub: [ 1.    1.    0.76159    0.76159]
```



plot_star(input)



plot_star(output)

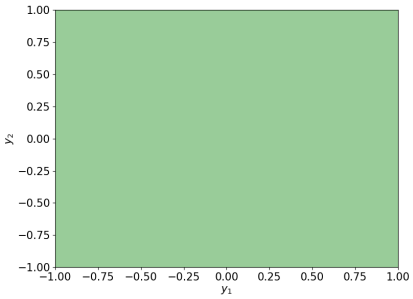
Hyperbolic Tangent (TanH) layer over-approximate reachability analysis with **SparseStar**.

```
# Construct a TanH layer
L_tanh = TanSigLayer()

# Construct input set
lb = np.array([-1.0, -1.0])
```

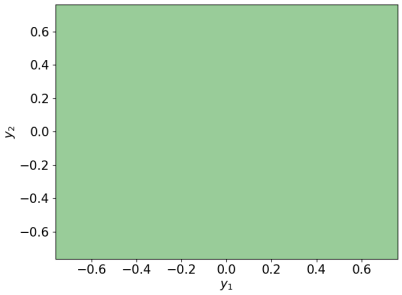
```
Input set:
SparseStar Set:
A:
[[0. 1. 0.]
 [0. 0. 1.]]
C_csc:
```

```
ub = np.array([1.0, 1.0])
input = SparseStar(lb, ub)
print('\nInput set:')
print(input)
plot_star(input)

# Reachability analysis
output = L_tanh.reach(input, method='approx')
print('\nOutput set:')
print(output)
plot_star(output)
```

```
[]
d: []
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
pred_depth: [0. 0.]
dim: 2
nVars: 2
nZVars: 0
nIVars: 2

Output set:
SparseStar Set:
A:
[[0. 1. 0.]
 [0. 0. 1.]]
C_csc:
[[ 0.41997  0.      -1.       0.     ]
 [ 0.       0.41997  0.      -1.     ]
 [-0.41997  0.       1.       0.     ]
 [ 0.      -0.41997  0.       1.     ]
 [ 0.84996  0.      -1.       0.     ]
 [ 0.       0.84996  0.      -1.     ]
 [-0.84996  0.       1.       0.     ]
 [ 0.      -0.84996  0.       1.     ]]
d: [0.34162 0.34162 0.34162 0.34162 0.08837
↪  0.08837 0.08837 0.08837]
pred_lb: [-1.   -1.   -0.76159  -0.76159]
pred_ub: [ 1.    1.    0.76159   0.76159]
pred_depth: [1. 1. 0. 0.]
dim: 2
nVars: 4
nZVars: 2
nIVars: 2
```



plot_star(input)



plot_star(output)

8 Convolution Layer

Mathematically, the convolution operation with the weight matrix $W_{conv} \in \mathbb{R}^{p\times q\times ch_{in}\times ch_{out}}$ and input $X \in \mathbb{R}^{h\times w\times ch_{in}\times m}$ is

$$Y[s,t,u,m] = \sum_{k=1}^{ch_{in}} \sum_{i=1}^{p} \sum_{j=1}^{q} W_{conv}[i,j,k,u] X[i+s, j+t, k, m]. \tag{4.1}$$

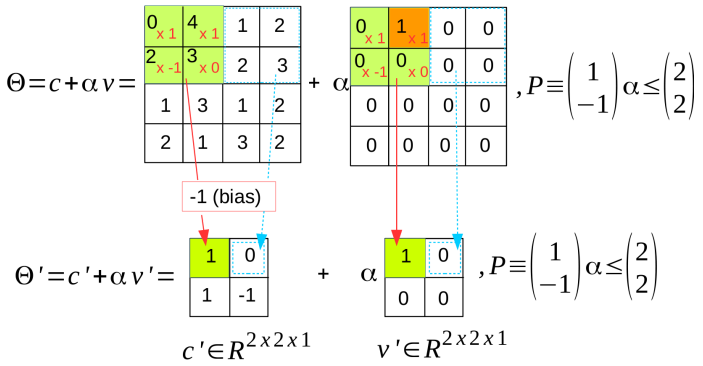


Figure 4.3: Rechability of convolutional layer using ImageStar.

Lemma 4. *Given a convolution layer* $L_{conv} = \langle W_{conv} \in \mathbb{R}^{p\times q\times ch_{in}\times ch_{out}}, b_{conv} \in \mathbb{R}^{ch_{out}}, padding, stride, dilation\rangle$ *with an input set* $\mathcal{I} = \langle c, V, P\rangle \in \{ImageStar,\ SparseImageStar\}$*, the reachability of a convolution operation* ($\circledast$) *creates another reachable set* $\mathcal{R} = \langle c', V', P'\rangle \in \{ImageStar, SparseImageStar\}$ *with the following characteristics,*

$$c' = W_c \circledast c + b_c, V' = W_c \circledast V, P' \equiv P.$$

Example 3 (Reachable set of a convolution layer)**.** *The reachable set of a convolution layer with* 2×2 *kernel and the ImageStar input set is described in Figure 4.3, where the weights and the bias of the kernel are* $W_c = \begin{bmatrix} 1 & 1 \\ -1 & 0 \end{bmatrix}$ *and* $b = -1$*, respectively, the stride is* $S = [2,2]$*, the padding is* $P = [0,0,0,0]$*, and the dilation factor is* $D = [1,1]$*.*

8.1 Convolution Layer Construction

Construct a convolution layer (Conv2DLayer) with specified weight and bias.

```
W = np.array([[1, 1], [-1, 0]])
b = np.array([-1])

layer = [W, b]
stride = [2, 2]
padding = [0, 0, 0, 0]
dilation = [1, 1]
L_conv = Conv2DLayer(layer, stride, padding, dilation)
print(L_conv)
```

```
# print(L_conv)
Convolutional 2D Layer
module: default
in_channel: 1
out_channel: 1
stride: [2 2]
padding: [0 0]
dilation: [1 1]
sparse: False
weight: (2, 2, 1, 1), float64
bias: (1,), float64
```

Construct a convolution layer using a torch layer

```
layer = torch.nn.Conv2d(2, 3, (3, 3), stride=(2, 2), padding=(2,
↪  2))
L_conv = Conv2DLayer(layer)
print(L_conv)
```

```
# print(L_conv)
Convolutional 2D Layer
module: default
in_channel: 3
out_channel: 3
stride: [2 2]
padding: [2 2 2 2]
dilation: [1 1]
sparse: False
weight: (3, 3, 2, 3), float64
bias: (3,), float64
```

Construct a convolution layer with random weight and bias

```
L_conv = Conv2DLayer.rand(3, 3, 4, 5)
L_conv.info()
```

```
# L_conv.info()
Convolutional 2D Layer
module: default
in_channel: 4
out_channel: 5
stride: [1 1]
padding: [0 0]
dilation: [1 1]
sparse: False
weight: (3, 3, 4, 5), float64
bias: (5,), float64
```

8.2 Convolution Layer Reachability Analysis

Reachability analysis on convolution layer using **ImageStar** set illustrating Example 3.

```
W = np.array([[1, 1], [-1, 0]])
b = np.array([-1])

layer = [W, b]
stride = [2, 2]
padding = [0, 0, 0, 0]
dilation = [1, 1]
L_conv = Conv2DLayer(layer, stride,
↪  padding, dilation)
print(L_conv)

# Construct input set
h, w, ci, co = 4, 4, 1, 1
c = np.array([[0, 4, 1, 2],
              [2, 3, 2, 3],
              [1, 3, 1, 2],
              [2, 1, 3, 2]]
            ).reshape(h, w, ci, co)
v = np.zeros([h, w, ci, co])
v[0, 1, 0, 0] = 1

V = np.concatenate([c, v], axis=3)
C = np.array([[1], [-1]])
d = 2*np.ones([2])
pred_lb = -2*np.ones(1)
pred_ub = 2*np.ones(1)
```

```
# print(L_conv)
Convolutional 2D Layer
module: default
in_channel: 1
out_channel: 1
stride: [2 2]
padding: [0 0]
dilation: [1 1]
sparse: False
weight: (2, 2, 1, 1), float64
bias: (1,), float64

# print(IM)
Input ImageStar set:
 ImageStar Set:
V:
[[[[0. 0.]]
  [[4. 1.]]
  [[1. 0.]]
  [[2. 0.]]]
 [[[2. 0.]]
  [[3. 0.]]
  [[2. 0.]]
  [[3. 0.]]]
 [[[1. 0.]]
  [[3. 0.]]
  [[1. 0.]]
  [[2. 0.]]]
 [[[2. 0.]]
  [[1. 0.]]
  [[3. 0.]]
  [[2. 0.]]]]
C:
```

```
# print(R)
Output ImageStar set:
 ImageStar Set:
V:
[[[[ 1.  1.]]
  [[ 0.  0.]]]
 [[[ 1.  0.]]
  [[-1.  0.]]]]
C:
[[ 1]
 [-1]]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
height: 2
width: 2
num_channel: 1
num_pred: 1
```

```
IM = ImageStar(V, C, d, pred_lb, pred_ub)
print('Input ImageStar set:\n', IM)

R = L_conv.reach(IM)
print('Output ImageStar set:\n', R)
```

```
[[ 1]
 [-1]]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
height: 4
width: 4
num_channel: 1
num_pred: 1
```

Reachability analysis on convolution layer using **SparseImageStar (CSR & COO)** set illustrating Example 3.

```
W = np.array([[1, 1], [-1, 0]])
b = np.array([-1])

layer = [W, b]
stride = [2, 2]
padding = [0, 0, 0, 0]
dilation = [1, 1]
L_conv = Conv2DLayer(layer, stride, padding, dilation)
print(L_conv)

# Construct input set
h, w, ci, co = 4, 4, 1, 1
shape = (h, w, ci)

c = np.array([[0, 4, 1, 2],
              [2, 3, 2, 3],
              [1, 3, 1, 2],
              [2, 1, 3, 2]]).ravel()
V = np.zeros([h, w, ci, co])
V[0, 1, 0, 0] = 1
V = V.reshape(-1, 1)
V_csr = sp.csr_array(V)
V_coo = sp.coo_array(V)

C = np.array([[1], [-1]])
C = sp.csr_array(C)
d = 2*np.ones([2])
pred_lb = -2*np.ones(1)
pred_ub = 2*np.ones(1)

SIM_coo = SparseImageStar2DCOO(c, V_coo, C, d, pred_lb, pred_ub, shape)
print('Input SparseImageStar COO set:\n', SIM_coo)

R_coo = L_conv.reach(SIM_coo)
print('Output SparseImageStar COO set:\n', R_coo)

SIM_csr = SparseImageStar2DCSR(c, V_csr, C, d, pred_lb, pred_ub, shape)
print('Input SparseImageStar CSR set:\n', SIM_csr)

R_csr = L_conv.reach(SIM_csr)
print('Output SparseImageStar CSR set:\n', R_csr)
```

```
# print(L_conv)
Convolutional 2D Layer
module: default
in_channel: 1
out_channel: 1
stride: [2 2]
padding: [0 0]
dilation: [1 1]
sparse: False
weight: (2, 2, 1, 1), float64
bias: (1,), float64

Input SparseImageStar COO set:
 SparseImageStar2DCOO Set:
c: [0 4 1 2 2 3 2 3 1 3 1 2 2 1 3 2]
V_coo:
  (1, 0)        1.0
C_csr:
   data: [ 1 -1]
indices: [0 0]
 indptr: [0 1 2]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
shape: (4, 4, 1)
num_pred: 1
density: 0.0625
nnz: 1

Output SparseImageStar COO set:
 SparseImageStar2DCOO Set:
c: [ 1.  0.  1. -1.]
V_coo:
  (0, 0)        1.0
C_csr:
   data: [ 1 -1]
indices: [0 0]
 indptr: [0 1 2]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
shape: (2, 2, 1)
num_pred: 1
density: 0.25
nnz: 1

Input SparseImageStar CSR set:
 SparseImageStar2DCSR Set:
c: [0 4 1 2 2 3 2 3 1 3 1 2 2 1 3 2]
V_csr:
   data: [1.]
indices: [0]
 indptr: [0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
C_csr:
   data: [ 1 -1]
indices: [0 0]
 indptr: [0 1 2]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
shape: (4, 4, 1)
num_pred: 1
density: 0.0625
nnz: 1

Output SparseImageStar CSR set:
 SparseImageStar2DCSR Set:
c: [ 1.  0.  1. -1.]
V_csr:
   data: [1.]
indices: [0]
 indptr: [0 1 1 1 1]
C_csr:
   data: [ 1 -1]
indices: [0 0]
 indptr: [0 1 2]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
shape: (2, 2, 1)
num_pred: 1
density: 0.25
nnz: 1
```

9 Average Pooling Layer

The average pooling operation is mathematically defined as

$$Y[s,t,u,m] = \frac{1}{p*q}\sum_{i=1}^{p}\sum_{j=1}^{q} X[s*stride[0]+i, t*stride[1]+j, u, m].$$

Lemma 5. *Given an average pooling layer* $L_{avg} = \langle kenel\_size, padding, stride \rangle$ *with an input set* $\mathcal{I} = \langle c, V, P \rangle \in \{ImageStar, SparseImageStar\}$*, the reachability of an average pooling operation* ($\mathtt{avg}(\cdot)$) *creates another reachable set* $\mathcal{R} = \langle c', V', P' \rangle \in \{ImageStar, SparseImageStar\}$ *with the following characteristics,*

$$c' = \mathtt{avg}(c), V' = \mathtt{avg}(V), P' \equiv P.$$

Example 4 (Reachable set of an average pooling layer)**.** *The reachable set of a* 2×2 *average pooling layer with padding size* $P = [0, 0, 0, 0]$*, stride* $S = [2, 2]$ *and an ImageStar input is shown in Figure 4.4.*

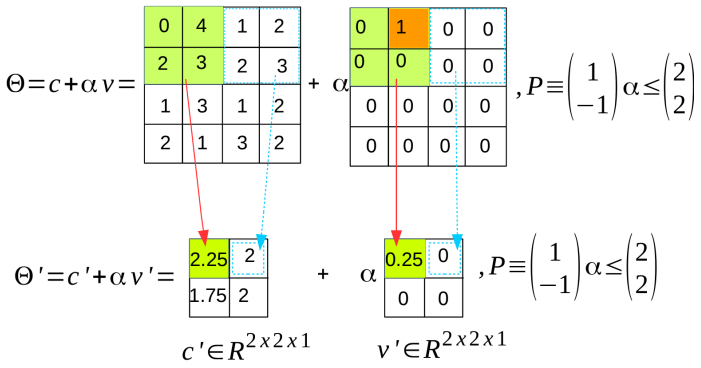


Figure 4.4: Rechability of average pooling layer using ImageStar.

9.1 Average Pooling Layer Construction

Construct an average pooling layer (AvgPool2DLayer)

```
kernel_size = [2, 2]
stride = [2, 2]
padding = [0, 0, 0, 0]
L_avg = AvgPool2DLayer(kernel_size, stride, padding)
print(L_avg)
```

```
# print(L_avg)
Average Pooling 2D Layer
module: default
kernel size: [2 2]
stride: [2 2]
padding: [0 0]
```

9.2 Average Pooling Layer Reachability Analysis

Reachability analysis on average pooling layer using **ImageStar** set illustration Example 4.

```
kernel_size = [2, 2]
stride = [2, 2]
padding = [0, 0, 0, 0]
L_avg = AvgPool2DLayer(kernel_size,
↪  stride, padding)

# Construct input set
h, w, ci, m = 4, 4, 1, 1
c = np.array([[0, 4, 1, 2],
              [2, 3, 2, 3],
              [1, 3, 1, 2],
              [2, 1, 3, 2]]
            ).reshape(h, w, ci, 1)
v = np.zeros([h, w, ci, m])
v[0, 1, 0, 0] = 1

V = np.concatenate([c, v], axis=3)
C = np.array([[1], [-1]])
d = 2*np.ones([2])
pred_lb = -2*np.ones(1)
pred_ub = 2*np.ones(1)

IM = ImageStar(V, C, d, pred_lb, pred_ub)
print('Input ImageStar set:\n', IM)
```

```
Input ImageStar set:
ImageStar Set:
V:
[[[[0. 0.]]
  [[4. 1.]]
  [[1. 0.]]
  [[2. 0.]]]
 [[[2. 0.]]
  [[3. 0.]]
  [[2. 0.]]
  [[3. 0.]]]
 [[[1. 0.]]
  [[3. 0.]]
  [[1. 0.]]
  [[2. 0.]]]
 [[[2. 0.]]
  [[1. 0.]]
  [[3. 0.]]
  [[2. 0.]]]]
C:
[[ 1]
 [-1]]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
height: 4
width: 4
num_channel: 1
num_pred: 1
```

```
Output ImageStar set:
ImageStar Set:
V:
[[[[2.25 0.25]]
   [[2.   0.  ]]]
 [[[1.75 0.  ]]
   [[2.   0.  ]]]]
C:
[[ 1]
 [-1]]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
height: 2
width: 2
num_channel: 1
num_pred: 1
```

```

R = L_avg.reach(IM)
print('Output ImageStar set:\n', R)
```

Reachability analysis on average pooling layer using **SparseImageStar (CSR & COO)** set illustration Example 4.

```
kernel_size = [2, 2]
stride = [2, 2]
padding = [0, 0, 0, 0]
L_avg = AvgPool2DLayer(kernel_size, stride, padding)

# Construct input set
h, w, ci, m = 4, 4, 1, 1
shape = (h, w, ci)

c = np.array([[0, 4, 1, 2],
              [2, 3, 2, 3],
              [1, 3, 1, 2],
              [2, 1, 3, 2]]
            ).ravel()
V = np.zeros([h, w, ci, m])
V[0, 1, 0, 0] = 1
V = V.reshape(-1, 1)
V_csr = sp.csr_array(V)
V_coo = sp.coo_array(V)

C = np.array([[1], [-1]])
C = sp.csr_array(C)
d = 2*np.ones([2])
pred_lb = -2*np.ones(1)
pred_ub = 2*np.ones(1)

SIM_coo = SparseImageStar2DCOO(c, V_coo, C, d, pred_lb, pred_ub, shape)
print('Input SparseImageStar COO set:\n', SIM_coo)

R_coo = L_avg.reach(SIM_coo)
print('Output SparseImageStar COO set:\n', R_coo)

SIM_csr = SparseImageStar2DCSR(c, V_csr, C, d, pred_lb, pred_ub, shape)
print('Input SparseImageStar CSR set:\n', SIM_csr)

R_csr = L_avg.reach(SIM_csr)
print('Output SparseImageStar CSR set:\n', R_csr)
```

```
Input SparseImageStar COO set:
 SparseImageStar2DCOO Set:
c: [0 4 1 2 2 3 2 3 1 3 1 2 2 1 3 2]
V_coo:
  (1, 0)        1.0
C_csr:
   data: [ 1 -1]
indices: [0 0]
 indptr: [0 1 2]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
shape: (4, 4, 1)
num_pred: 1
density: 0.0625
nnz: 1


Output SparseImageStar COO set:
 SparseImageStar2DCOO Set:
c: [2.25 2.   1.75 2.  ]
V_coo:
  (0, 0)        0.25
C_csr:
   data: [ 1 -1]
indices: [0 0]
 indptr: [0 1 2]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
shape: (2, 2, 1)
num_pred: 1
density: 0.25
nnz: 1
```

```
Input SparseImageStar CSR set:
 SparseImageStar2DCSR Set:
c: [0 4 1 2 2 3 2 3 1 3 1 2 2 1 3 2]
V_csr:
   data: [1.]
indices: [0]
 indptr: [0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
C_csr:
   data: [ 1 -1]
indices: [0 0]
 indptr: [0 1 2]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
shape: (4, 4, 1)
num_pred: 1
density: 0.0625
nnz: 1


Output SparseImageStar CSR set:
 SparseImageStar2DCSR Set:
c: [2.25 2.   1.75 2.  ]
V_csr:
   data: [0.25]
indices: [0]
 indptr: [0 1 1 1 1]
C_csr:
   data: [ 1 -1]
indices: [0 0]
 indptr: [0 1 2]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
shape: (2, 2, 1)
num_pred: 1
density: 0.25
nnz: 1
```

10 Batch Normalization Layer

The batch normalization operation initially normalizes the input images with respect to channel dimension, i.e. (ch), with ϵ, a value added to the denominator for numerical stability, and the trainable parameters: mean μ and standard deviation σ^2. Then, it affine-maps the input with the learnable parameters: γ and β. The mathematical formulation of the batch normalization is shown below:

$$y^{(ch)} = \gamma^{(ch)} \frac{x^{(ch)} - \mu^{(ch)}}{\sqrt{(\sigma^{(ch)})^2 + \epsilon}} + \beta^{(ch)}.$$

Lemma 6. *Given a batch normalization layer* $L_{bn} = \langle \mu \in \mathbb{R}^{ch}, \sigma^2 \in \mathbb{R}^{ch}, \epsilon \in \mathbb{R}^1, \gamma \in \mathbb{R}^{ch}, \beta \in \mathbb{R}^{ch} \rangle$ *with an input set* $\mathcal{I} = \langle c, V, P \rangle \in \{$*ImageStar, SparseImageStar*$\}$*, the reachability of a batch normalization operation creates another*

reachable set $\mathcal{R} = \langle c', V', P' \rangle \in \{ImageStar, SparseImageStar\}$ *with the following characteristics,*

$$c' = \gamma \frac{c - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta, V' = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} V, P' \equiv P.$$

10.1 Batch Normalization Layer Construction

Construct a batch normalization layer (BatchNorm2DLayer)

```
in_channels = 3
gamma = np.array([0.2, 0.7, 0.6])
beta = np.array([0.01, 0.2, 0.4])
mean = np.zeros(in_channels)
var = np.ones(in_channels)
layer = [gamma, beta, mean, var]
L_bn = BatchNorm2DLayer(layer, eps=2e-05)
print(L_bn)
```

```
Batch Normalization 2D Layer
module: default
gamma: [0.2 0.7 0.6]
beta: [0.01 0.2  0.4 ]
num_features: 3
epsilon: 2e-05
mean: [0. 0. 0.]
variance: [1. 1. 1.]
```

Construct a batch normalization layer (BatchNorm2DLayer) with Pytorch BatchNorm2D layer

```
in_channels = 3
layer = torch.nn.BatchNorm2d(in_channels)
L_bn = BatchNorm2DLayer(layer)
print(L_bn)
```

```
# print(L_bn)
Batch Normalization 2D Layer
module: default
gamma: [1. 1. 1.]
beta: [0. 0. 0.]
num_features: 3
epsilon: 1e-05
mean: [0. 0. 0.]
variance: [1. 1. 1.]
```

10.2 Batch Normalization Layer Reachability Analysis

Reachability analysis on batch normalization layer using **ImageStar**.

```
h, w = 2, 2
in_channels = 2
shape = (h, w, in_channels)

gamma = np.array([0.2, 0.7])
beta = np.array([0.01, 0.4])
mean = np.zeros(in_channels)
var = np.ones(in_channels)
layer = [gamma, beta, mean, var]
L_bn = BatchNorm2DLayer(layer, eps=2e-05)
print(L_bn)

lb = -np.arange(np.prod(shape))/10
ub = np.arange(np.prod(shape))[::-1]/5

lb = lb.reshape(h, w, in_channels)
ub = ub.reshape(h, w, in_channels)
IM = ImageStar(lb, ub)
print('Input ImageStar set:\n', IM)

R = L_bn.reach(IM)
print('Output ImageStar set:\n', R)
```

```
# print(L_bn)
Batch Normalization 2D Layer
module: default
gamma: [0.2 0.7]
beta: [0.01 0.4 ]
num_features: 2
epsilon: 2e-05
mean: [0. 0.]
variance: [1. 1.]

Input ImageStar set:
 ImageStar Set:
V:
[[[[ 0.7   0.7   0.    0.    0.    0.    0.    0.    0.  ]
   [ 0.55  0.    0.65  0.    0.    0.    0.    0.    0.  ]]
  [[ 0.4   0.    0.    0.6   0.    0.    0.    0.    0.  ]
   [ 0.25  0.    0.    0.    0.55  0.    0.    0.    0.  ]]]
 [[[ 0.1   0.    0.    0.    0.    0.5   0.    0.    0.  ]
   [-0.05  0.    0.    0.    0.    0.    0.45  0.    0.  ]]
  [[-0.2   0.    0.    0.    0.    0.    0.    0.4   0.  ]
   [-0.35  0.    0.    0.    0.    0.    0.    0.    0.35]]]]
C:
[]
d: []
pred_lb: [-1. -1. -1. -1. -1. -1. -1. -1.]
pred_ub: [1. 1. 1. 1. 1. 1. 1. 1.]
height: 2
width: 2
num_channel: 2
num_pred: 8

Output ImageStar set:
 ImageStar Set:
V:
[[[[ 0.15  0.14  0.    0.    0.    0.    0.    0.    0.   ]
   [ 0.785 0.    0.455 0.    0.    0.    0.    0.    0.   ]]
  [[ 0.09  0.    0.    0.12  0.    0.    0.    0.    0.   ]
   [ 0.575 0.    0.    0.    0.385 0.    0.    0.    0.   ]]]
 [[[ 0.03  0.    0.    0.    0.    0.1   0.    0.    0.   ]
   [ 0.365 0.    0.    0.    0.    0.    0.315 0.    0.   ]]
  [[-0.03  0.    0.    0.    0.    0.    0.    0.08  0.   ]
   [ 0.155 0.    0.    0.    0.    0.    0.    0.    0.245]]]]
C:
[]
```

```
d: []
pred_lb: [-1. -1. -1. -1. -1. -1. -1. -1.]
pred_ub: [1. 1. 1. 1. 1. 1. 1. 1.]
height: 2
width: 2
num_channel: 2
num_pred: 8
```

Reachability analysis on batch normalization layer using **SparseImageStar (CSR & COO)**.

```
h, w = 2, 2
in_channels = 2
shape = (h, w, in_channels)

gamma = np.array([0.2, 0.7])
beta = np.array([0.01, 0.4])
mean = np.zeros(in_channels)
var = np.ones(in_channels)
layer = [gamma, beta, mean, var]
L_bn = BatchNorm2DLayer(layer, eps=2e-05)
print(L_bn)

lb = -np.arange(np.prod(shape))/10
ub = np.arange(np.prod(shape))[::-1]/5

lb = lb.reshape(h, w, in_channels)
ub = ub.reshape(h, w, in_channels)

SIM_coo = SparseImageStar2DCOO(lb, ub)
print('Input SparseImageStar COO set:\n', SIM_coo)

R_coo = L_bn.reach(SIM_coo)
print('Output SparseImageStar COO set:\n', R_coo)

SIM_csr = SparseImageStar2DCSR(lb, ub)
print('Input SparseImageStar CSR set:\n', SIM_csr)

R_csr = L_bn.reach(SIM_csr)
print('Output SparseImageStar CSR set:\n', R_csr)
```

```
# print(L_bn)
Batch Normalization 2D Layer
module: default
gamma: [0.2 0.7]
beta: [0.01 0.4 ]
num_features: 2
epsilon: 2e-05
mean: [0. 0.]
variance: [1. 1.]

Input SparseImageStar COO set:
 SparseImageStar2DCOO Set:
c: [ 0.7   0.55  0.4   0.25  0.1  -0.05 -0.2
↪  -0.35]
V_coo:
  (0, 0)        0.7
  (1, 1)        0.65
  (2, 2)        0.6
  (3, 3)        0.55
  (4, 4)        0.5
  (5, 5)        0.45
  (6, 6)        0.4
  (7, 7)        0.35
C_csr:
   data: []
indices: []
 indptr: [0]
d: []
pred_lb: [-1. -1. -1. -1. -1. -1. -1. -1.]
pred_ub: [1. 1. 1. 1. 1. 1. 1. 1.]
shape: (2, 2, 2)
num_pred: 8
density: 0.125
nnz: 8

Output SparseImageStar COO set:
 SparseImageStar2DCOO Set:
c: [ 0.15   0.785  0.09   0.575  0.03
↪  0.365 -0.03   0.155]
V_coo:
  (0, 0)        0.13999860002099965
  (1, 1)        0.4549954500682489
  (2, 2)        0.1199988000179997
  (3, 3)        0.38499615005774906
  (4, 4)        0.09999900001499976
  (5, 5)        0.3149968500472492
  (6, 6)        0.07999920001199981
  (7, 7)        0.24499755003674936
C_csr:
   data: []
indices: []
 indptr: [0]
d: []
pred_lb: [-1. -1. -1. -1. -1. -1. -1. -1.]
pred_ub: [1. 1. 1. 1. 1. 1. 1. 1.]
shape: (2, 2, 2)
num_pred: 8
density: 0.125
nnz: 8
```

```
Input SparseImageStar CSR set:
 SparseImageStar2DCSR Set:
c: [ 0.7   0.55  0.4   0.25  0.1  -0.05 -0.2
↪  -0.35]
V_csr:
   data: [0.7  0.65 0.6  0.55 0.5  0.45 0.4
   ↪  0.35]
indices: [0 1 2 3 4 5 6 7]
 indptr: [0 1 2 3 4 5 6 7 8]
C_csr:
   data: []
indices: []
 indptr: [0]
d: []
pred_lb: [-1. -1. -1. -1. -1. -1. -1. -1.]
pred_ub: [1. 1. 1. 1. 1. 1. 1. 1.]
shape: (2, 2, 2)
num_pred: 8
density: 0.125
nnz: 8


Output SparseImageStar CSR set:
 SparseImageStar2DCSR Set:
c: [ 0.15   0.785  0.09   0.575  0.03
↪  0.365 -0.03   0.155]
V_csr:
   data: [0.14  0.455 0.12  0.385 0.1
   ↪  0.315 0.08  0.245]
indices: [0 1 2 3 4 5 6 7]
 indptr: [0 1 2 3 4 5 6 7 8]
C_csr:
   data: []
indices: []
 indptr: [0]
d: []
pred_lb: [-1. -1. -1. -1. -1. -1. -1. -1.]
pred_ub: [1. 1. 1. 1. 1. 1. 1. 1.]
shape: (2, 2, 2)
num_pred: 8
density: 0.125
nnz: 8
```

11 Max Pooling Layer

The mathematical representation of max pooling is described below.

$$Y[s,t,u,m] = \max_{i=1,\cdots,p} \max_{j=1,\cdots,q} X[s * stride[0] + i, t * stride[1] + j, u, m]$$

Example 5 (Reachable set of a max pooling layer)**.** *The reachable set of a* 2×2 *max pooling layer with padding size* $P = [0,0,0,0]$*, stride* $S = [2,2]$ *and an ImageStar input is shown in Figure 4.5.*

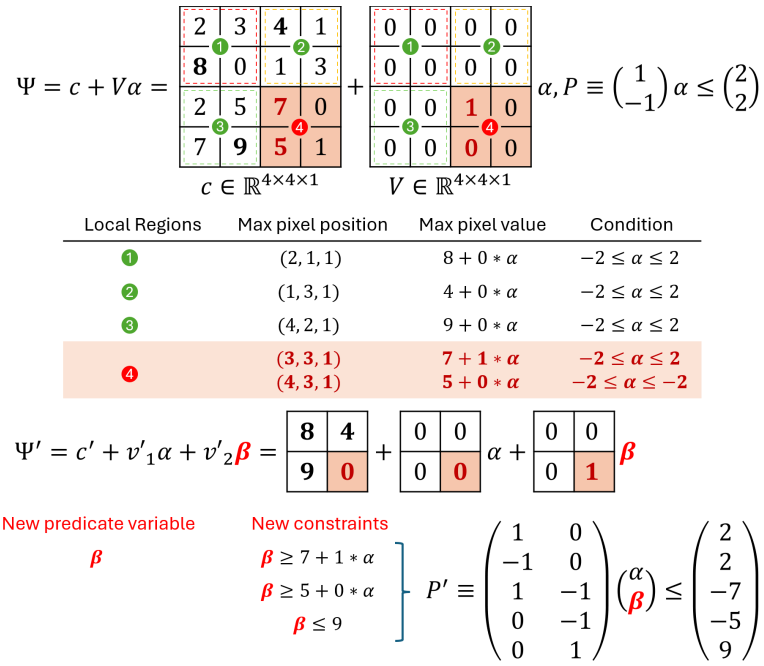


Figure 4.5: Over-approximate reachability analysis on max pooling layer with 2×2 kernel size, $(2, 2)$ strides, and no padding.

11.1 Max Pooling Layer Construction

Construct a max pooling layer (MaxPool2DLayer)

```
kernel_size = [2, 2]
stride = [2, 2]
padding = [0, 0, 0, 0]
L_max = MaxPool2DLayer(kernel_size, stride, padding)
print(L_max)
```

```
# print(L_max)
Max Pooling 2D Layer
module: default
kernel size: [2 2]
stride: [2 2]
padding: [0 0]
```

11.2 Max Pooling Layer Reachability Analysis

Reachability analysis on max pooling layer using **ImageStar** illustrating Example 5.

```
kernel_size = [2, 2]
stride = [2, 2]
padding = [0, 0, 0, 0]
L_max = MaxPool2DLayer(kernel_size,
↪  stride, padding)

# Construct input set
h, w, ci, m = 4, 4, 1, 1
c = np.array([[2, 3, 4, 1], [8, 0, 1, 3],
↪  [2, 5, 7, 0], [7, 9, 6,
↪  1]]).reshape(h, w, ci, m)
v = np.zeros([h, w, ci, m])
v[2, 2, 0, 0] = 1

V = np.concatenate([c, v], axis=3)
C = np.array([[1], [-1]])
d = 2*np.ones([2])
pred_lb = -2*np.ones(1)
pred_ub = 2*np.ones(1)

IM = ImageStar(V, C, d, pred_lb, pred_ub)
print('Input ImageStar set:\n', IM)

R = L_max.reach(IM, 'approx')
```

```
Input ImageStar set:
 ImageStar Set:
V:
[[[[2. 0.]]
  [[3. 0.]]
  [[4. 0.]]
  [[1. 0.]]]
 [[[8. 0.]]
  [[0. 0.]]
  [[1. 0.]]
  [[3. 0.]]]
 [[[2. 0.]]
  [[5. 0.]]
  [[7. 1.]]
  [[0. 0.]]]
 [[[7. 0.]]
  [[9. 0.]]
  [[6. 0.]]
  [[1. 0.]]]]
C:
[[ 1]
 [-1]]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
height: 4
width: 4
num_channel: 1
num_pred: 1
```

```
Output ImageStar set:
 ImageStar Set:
V:
[[[[8. 0.]]
  [[4. 0.]]]
 [[[9. 0.]]
  [[7. 1.]]]]
C:
[[ 1]
 [-1]]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
height: 2
width: 2
num_channel: 1
num_pred: 1
```

```
print('Output ImageStar set:\n', R)
```

Reachability analysis on max pooling layer using **SparseImageStar (CSR & COO)**.

```
kernel_size = [2, 2]
stride = [2, 2]
padding = [0, 0, 0, 0]
L_max = MaxPool2DLayer(kernel_size, stride, padding)

# Construct input set
h, w, ci, m = 4, 4, 1, 1
shape = (h, w, ci)

c = np.array([[2, 3, 4, 1], [8, 0, 1, 3], [2, 5, 7, 0], [7, 9, 5, 1]]).ravel()
V = np.zeros([h, w, ci, m])
V[2, 2, 0, 0] = 1
V = V.reshape(-1, 1)
V_csr = sp.csr_array(V)
V_coo = sp.coo_array(V)

C = np.array([[1], [-1]])
C = sp.csr_array(C)
d = 2*np.ones([2])
pred_lb = -2*np.ones(1)
pred_ub = 2*np.ones(1)

SIM_coo = SparseImageStar2DCOO(c, V_coo, C, d, pred_lb, pred_ub, shape)
print('Input SparseImageStar COO set:\n', SIM_coo)

R_coo = L_max.reach(SIM_coo, 'approx')
print('Output SparseImageStar COO set:\n', R_coo)

SIM_csr = SparseImageStar2DCSR(c, V_csr, C, d, pred_lb, pred_ub, shape)
print('Input SparseImageStar CSR set:\n', SIM_csr)

R_csr = L_max.reach(SIM_csr, 'approx')
print('Output SparseImageStar CSR set:\n', R_csr)
```

```
Input SparseImageStar COO set:
 SparseImageStar2DCOO Set:
c: [2 3 4 1 8 0 1 3 2 5 7 0 7 9 5 1]
V_coo:
  (10, 0)       1.0
C_csr:
   data: [ 1 -1]
indices: [0 0]
 indptr: [0 1 2]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
shape: (4, 4, 1)
num_pred: 1
density: 0.0625
nnz: 1

Output SparseImageStar COO set:
 SparseImageStar2DCOO Set:
c: [8. 4. 9. 7.]
V_coo:
  (3, 0)        1.0
C_csr:
   data: [ 1 -1]
indices: [0 0]
 indptr: [0 1 2]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
shape: (2, 2, 1)
num_pred: 1
density: 0.25
nnz: 1
```

```
Input SparseImageStar CSR set:
 SparseImageStar2DCSR Set:
c: [2 3 4 1 8 0 1 3 2 5 7 0 7 9 5 1]
V_csr:
   data: [1.]
indices: [0]
 indptr: [0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1]
C_csr:
   data: [ 1 -1]
indices: [0 0]
 indptr: [0 1 2]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
shape: (4, 4, 1)
num_pred: 1
density: 0.0625
nnz: 1

Output SparseImageStar CSR set:
 SparseImageStar2DCSR Set:
c: [8. 4. 9. 7.]
V_csr:
   data: [1.]
indices: [0]
 indptr: [0 0 0 0 1]
C_csr:
   data: [ 1 -1]
indices: [0 0]
 indptr: [0 1 2]
d: [2. 2.]
pred_lb: [-2.]
pred_ub: [2.]
shape: (2, 2, 1)
num_pred: 1
density: 0.25
nnz: 1
```

Chapter 5

Verification of Deep Neural Networks (DNNs)

1 Verification of FeedForward Neural Networks (FFNN)

Using StarV for the verification of feedforward neural networks (FFNNs) consists of the following main steps:

- Constructing an FFNN object.
- Constructing an input set used to verify the network.
- Choosing a reachability analysis method
- Specifying a property for the network to verify.
- Verifying the network.

1.1 FeedForward Neural Networks Construction

There are several ways to construct an FFNN object. Users can choose to manually construct the FFNN object layer by layer or load the weights and biases from `.mat` file and then construct the FFNN object.

Method 1: Manually Constructing an FFNN object

This is suitable when users are familiar with all of the network's information, such as the weight matrices, bias vectors, and activation functions for each layer. An FFNN object can be constructed using an array of layer objects. In the following example, we construct an FFNN with one fully connected layer, one ReLU layer, and another fully connected layer.

```
# Construct a feedforward neural network
layers = []
W1 = np.array([[1.0, -2.0], [-1., 0.5], [1., 1.5]])
b1 = np.array([0.5, 1.0, -0.5])
layer1 = [W1, b1]
L1 = FullyConnectedLayer(layer1)
L2 = ReLULayer()
W3 = np.array([[-1.0, -1.0, 1.0], [2.0, 1.0, -0.5]])
b3 = np.array([-0.2, -1.0])
layer3 = [W3, b3]
L3 = FullyConnectedLayer(layer3)

layers.append(L1)
layers.append(L2)
layers.append(L3)

F = NeuralNetwork(layers, 'ffnn_tiny_network')
print(F)
```

```
# print(F)
=============NETWORK===============
Network type: ffnn_tiny_network
Input Dimension: 2
Output Dimension: 2
Number of Layers: 3
Layer types:
Layer 0: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (3, 2,
↪  dtype=float64)
Layer 1: <class 'StarV.layer.ReLULayer.ReLULayer'>
Layer 2: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (2, 3,
↪  dtype=float64)
```

Method 2: Constructing an FFNN object from .mat File

Users can choose to load the network's weights and biases from `.mat` file and then construct the neural network object.

```
# Construct a feedforward neural network from a .mat file
starv_root_path = os.path.dirname(StarV.__file__)
net_path = starv_root_path +
↪  '/util/data/nets/ACASXU/ACASXU_run2a_1_1_batch_2000.mat'
mat_contents = loadmat(net_path)
W = mat_contents['W']
b = mat_contents['b']

layers = []
for i in range(0, b.shape[1]-1):
    Wi = W[0, i]
    bi = b[0, i]
    bi = bi.reshape(bi.shape[0],)
    L1 = FullyConnectedLayer([Wi, bi])
    layers.append(L1)
    L2 = ReLULayer()
    layers.append(L2)

Wi = W[0, b.shape[1]-1]
bi = b[0, b.shape[1]-1]
bi = bi.reshape(bi.shape[0],)
L1 = FullyConnectedLayer([Wi, bi])
layers.append(L1)
F = NeuralNetwork(layers, net_type='ffnn_ACASXU_1_1')
print(F)
```

```
# print(F)
=============NETWORK===============
Network type: ffnn_ACASXU_1_1
Input Dimension: 5
Output Dimension: 5
Number of Layers: 13
Layer types:
Layer 0: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (50, 5,
↪  dtype=float64)
Layer 1: <class 'StarV.layer.ReLULayer.ReLULayer'>
Layer 2: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (50, 50,
↪  dtype=float64)
Layer 3: <class 'StarV.layer.ReLULayer.ReLULayer'>
Layer 4: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (50, 50,
↪  dtype=float64)
Layer 5: <class 'StarV.layer.ReLULayer.ReLULayer'>
Layer 6: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (50, 50,
↪  dtype=float64)
Layer 7: <class 'StarV.layer.ReLULayer.ReLULayer'>
Layer 8: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (50, 50,
↪  dtype=float64)
Layer 9: <class 'StarV.layer.ReLULayer.ReLULayer'>
Layer 10: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (50, 50,
↪  dtype=float64)
Layer 11: <class 'StarV.layer.ReLULayer.ReLULayer'>
Layer 12: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (5, 50,
↪  dtype=float64)
```

1.2 Evaluate an Input Vector on an FFNN

We can evaluate an input vector on the network, as shown in the following example. (NOTE: We will use the same FFNN (`F, 'ffnn_tiny_network'`) to be created in this example throughout this section.)

```
# Construct a feedforward neural network
layers = []
W1 = np.array([[1.0, -2.0], [-1., 0.5], [1., 1.5]])
b1 = np.array([0.5, 1.0, -0.5])
layer1 = [W1, b1]
L1 = FullyConnectedLayer(layer1)
L2 = ReLULayer()
W2 = np.array([[-1.0, -1.0, 1.0], [2.0, 1.0, -0.5]])
b2 = np.array([-0.2, -1.0])
layer2 = [W2, b2]
L3 = FullyConnectedLayer(layer2)

layers.append(L1)
layers.append(L2)
layers.append(L3)

F = NeuralNetwork(layers, 'ffnn_tiny_network')
print(F)

x = np.array([-1.0, 2.0])
y = F.evaluate(x)
print("Input vector:", x)
print("Output after FFNN:", y)
```

```
# print(F)
=============NETWORK===============
Network type: ffnn_tiny_network
Input Dimension: 2
Output Dimension: 2
Number of Layers: 3
Layer types:
Layer 0: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (3, 2,
↪  dtype=float64)
Layer 1: <class 'StarV.layer.ReLULayer.ReLULayer'>
Layer 2: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (2, 3,
↪  dtype=float64)

Input vector: [-1.  2.]
Output after FFNN: [-1.7   1.25]
```

1.3 Qualitative Reachability Analysis of FFNN

To perform qualitative verification of a network, StarV computes the reachable set of the network layer-by-layer corresponding to a specific input set that is defined using Star set, i.e., the output of the current layer is the input for the next layer. StarV supports the exact reachability method for the network with piecewise linear activation functions

such as ReLU and SatLin, and the over-approximate reachability method (Lemma 2, Figure 4.1) for ReLU networks.

The number of cores utilized for computation is only of concern when using exact reachability methods. The over-approximate reachability method for ReLU networks uses one core for computation. In the exact analysis, a single input set can be split into multiple output sets after one layer. Therefore, to speed up the computation for the exact analysis, StarV leverages the Python `multiprocessing` library, 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 that they want to use for the computation, which depends on the configuration of their machines.

Definition 6 (Qualitative Reachability of FFNN)**.** *Given a bounded convex polytope input set $\mathcal{I} \triangleq \{x \mid Ax \leq b\}$, the reachability analysis of an FFNN $\mathcal{F} = \{L_i\}$, where $i \in [1, 2, \cdots, l]$, is analyzed layer-by-layer computing an output reachable set $\mathcal{R}_{\mathcal{F}}$ as follows,*

$$\begin{aligned}
\mathcal{R}_{L_1} &\triangleq \{y_1 \mid y_1 = L_1(x), x \in \mathcal{I}\},\\
\mathcal{R}_{L_2} &\triangleq \{y_2 \mid y_2 = L_2(y_1), y_1 \in \mathcal{R}_{L_1}\},\\
&\vdots\\
\mathcal{R}_{\mathcal{F}} = \mathcal{R}_{L_l} &\triangleq \{y_l \mid y_l = L_l(y_{l-1}), y_{l-1} \in \mathcal{R}_{L_{l-1}}\},
\end{aligned}$$

where $L_i \in \{L_{fc}, L_r, L_{lr}, L_{st}, L_{sts}, L_\sigma, L_\phi\}$ is the i^{th} layer operation.

The following example shows how to conduct exact and over-approximate reachability analysis for the FFNN using Star set. Users can choose to use parallel computing for the exact scheme but not for the over-approximation scheme, as only exactly one over-approximated reachable set will be propagated through the network layer by layer.

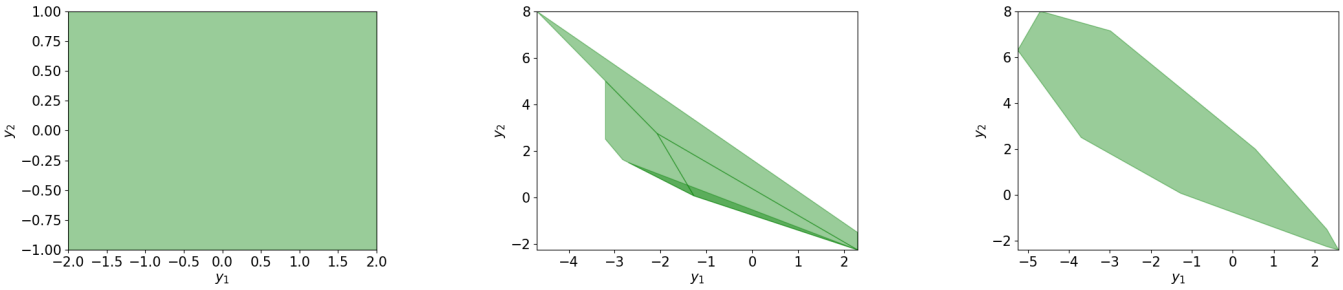
```
#  ... Construct a feedforward neural network ...

# Construct input set
lb = np.array([-2.0, -1.0])
ub = np.array([2.0, 1.0])
S = Star(lb, ub)
plot_star(S)

# Exact Reachability analysis
# ----- Parallelized version -----
numCores = 2
pool = multiprocessing.Pool(numCores)
# ----- Single-core version -----
# pool = None

I = [S]
R_exact = reachExactBFS(F, I, lp_solver='gurobi', pool=pool,
↪  show=True)
print('Number of Exact Reachable sets: ', len(R_exact))
plot_star(R_exact)

# Over-approximation reachability analysis
I = S
R_approx = reachApproxBFS(F, I, lp_solver='gurobi', show=True)
print('Number of Over-approximate Reachable sets: ',
↪  len(R_approx))
plot_star(R_approx)
```

```
-------------------------------------------
Exact Reachability Analysis:
-------------------------------------------
Computing layer 0 FullyConnectedLayer reachable set...
Number of stars/probstars: 1
Computing layer 1 ReLULayer reachable set...
Number of stars/probstars: 6
Computing layer 2 FullyConnectedLayer reachable set...
Number of stars/probstars: 6

Number of Exact Reachable sets:  6

-------------------------------------------
Over-approximate Reachability Analysis:
-------------------------------------------
Computing layer 0 FullyConnectedLayer reachable set...
Number of stars: 1
Computing layer 1 ReLULayer reachable set...
Number of stars: 1
Computing layer 2 FullyConnectedLayer reachable set...
Number of stars: 1

Number of Over-approximate Reachable sets:  1
```

1.4 Qualitative Verification of FFNN

After conducting a reachability analysis of the network, users need to specify the property that they want to verify with the network. The property is a linear predicate over the outputs of the network. Let $\mathcal{S}$ be a safe region. If the exact reachable sets of the network reach the unsafe region, the network is unsafe. Otherwise, it is safe. If the over-approximated reachable sets of the network reach the unsafe region, the result is unknown as we don't know if the intersection comes from the conservative errors. If the over-approximated reachable sets do not reach the unsafe region, the network is safe. The qualitative verification of FFNN is formally defined in Def. 7.

Definition 7 (Qualitative Safety Verification of a FFNN)**.** *Given a k-layers feed-forward neural network $\mathcal{F}$, and a*



(a) `plot_star(P).` (b) `plot_star(R_exact).` (c) `plot_star(R_approx).`

Figure 5.1: Visualization of Reachability Analysis on an FFNN with Star.

safety specification $\mathcal{S}$ defined as a set of linear constraints on the neural network outputs $S \triangleq \{y_k | Cy_k \leq d\}$, the neural network $\mathcal{F}$ is called to be safe corresponding to the input set $\mathcal{I}$, we write $\mathcal{F}(\mathcal{I}) \vDash S$, if and only if $R_{L_k} \cap \neg S = \emptyset$, where R_{L_k} is the reachable set of the neural network with the input set $\mathcal{I}$, and $\neg$ is the symbol for logical negation. Otherwise, the neural network is called to be unsafe $\mathcal{F}(\mathcal{I}) \nvDash S$.

There are two methods for performing qualitative verification of an FFNN. First, users could manually compute the output sets of the networks and then iteratively verify whether each output set satisfies the property. Otherwise, users could call the function provided in the StarV tool to verify the network automatically. Both examples are provided below.

Manually Verifying an FFNN

In this example, we want to manually verify whether the output reachable sets `R_exact`, `R_approx` that was computed previously of the network `F` violate the property of the network. Assume we want to verify the following property:

$$S = \{y \in \mathbb{R}^2 | y_1 \geq 4\} \tag{5.1}$$

To prove whether the network satisfies (SAT) or does not satisfy (UNSAT) property, we check the intersection between the reachable sets and the safety property. One can see that from the results when using the exact reachability methods. The output reachable sets do not reach the safety property. Therefore, StarV returns UNSAT for all reachable sets. When the over-approximate reachability method is used, StarV returns UNSAT as the result. We can confirm the results from the visualization of the output reachable sets for the network as shown in Figure 5.1.

When the network's reachable sets reach an unsafe region, we say the network is unsafe. Otherwise, it is safe. Using the over-approximate method, it is possible that the network is safe, but the conservative over-approximate reachable set obtained reaches the unsafe region. In this case, we cannot prove the network's safety using the over-approximate method.

```
# ... Construct a feedforward neural network ...

# ... Construct input set ...

# ... Exact reachability analysis ...

# ... Over-approximate reachability analysis ...

# Specifying a property of an FFNN
# y1 >= 4
unsafe_mat = np.array([[-1.0, 0]])
unsafe_vec = np.array([-4.0])

# Verification for R_exact
U_exact = [] # unsafe output set
Res_exact = [] # verification result
for R_exact_i in R_exact:
    U1 = checkSafetyStar(unsafe_mat, unsafe_vec, R_exact_i)
    if isinstance(U1, Star):
        U_exact.append(U1)
        Res_exact.append('SAT')
    else:
```

```
-------------------------------------------
Exact Qualitative Verification:
-------------------------------------------
Length of unsafe sets:  0
Verification result:  ['UNSAT', 'UNSAT', 'UNSAT',
                       'UNSAT', 'UNSAT', 'UNSAT']

-------------------------------------------
Over-approximate Qualitative Verification:
-------------------------------------------
Verification result:  UNSAT
```

```
        Res_exact.append('UNSAT')

print('length of unsafe sets: ', len(U_exact))
print('verification result: ', Res_exact)

# Verification for R_approx
U_approx = checkSafetyStar(unsafe_mat, unsafe_vec, R_approx)
if isinstance(U_approx, Star):
    Res_approx = 'UNKNOWN'
else:
    Res_approx = 'UNSAT'
print('Verification result: ', Res_approx)
```

Automatically Verifying an FFNN

In this example, we want to demonstrate how to verify an FFNN automatically using the supported functions in StarV. Assume we want to verify a different property:

$$S = \{y \in \mathbb{R}^2 | y_1 \leq -2\} \tag{5.2}$$

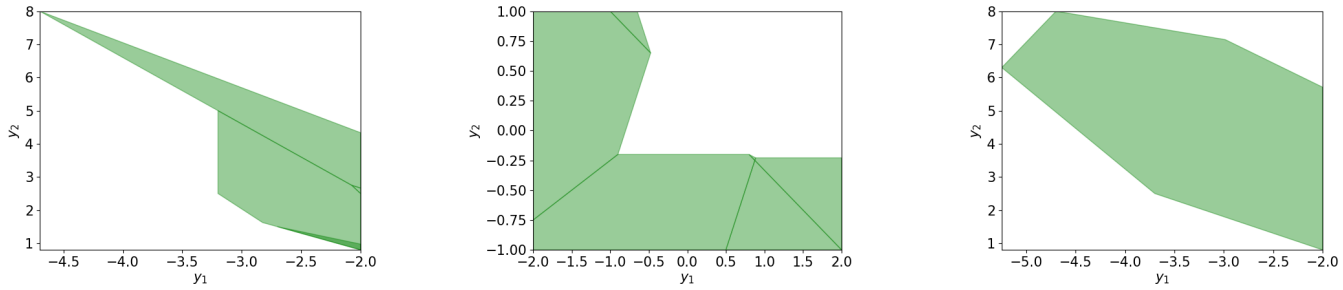
One can see from the results that the exact method can prove the network is unsafe. All of the output reachable sets of the network reach the unsafe region. Additionally, StarV can find the counterexample corresponding to reach unsafe output sets. A counterexample is a subset of the input that makes the network unsafe, i.e., the output of the network corresponding to the counter input relies on the unsafe region. The visualization of the results is shown in Figure 5.2a, 5.2b. For the over-approximate scheme, the reachable set reaches the unsafe region. However, we do not know if it is due to conservatives of the over-approximate reachable sets. Thus, StarV returns the result as UNKNOWN. The visualization of the result is shown in Figure. 5.2c.

```
# ... Construct a feedforward neural network ...

# ... Construct input set ...

# Specifying a property of an FFNN
# y1 <= -2
unsafe_mat = np.array([[1.0, 0]])
unsafe_vec = np.array([-2.0])

# Exact Verification
numCores = 2
I = [S]
R_exact, U_exact, C_exact, Res_exact = qualiVerifyExactBFS(F, I,
↪  unsafe_mat, unsafe_vec, numCores=numCores)
print('Number of Exact Reachable sets: ', len(R_exact))
print('Number of Exact Unsafe sets: ', len(U_exact))
print('Number of Counter Input sets: ', len(C_exact))
print('Verification result: ', Res_exact)
plot_star(R_exact)
plot_star(U_exact)
plot_star(C_exact)

# Over-approximate Verification
I = S
R_approx, U_approx, Res_approx = qualiVerifyApproxBFS(F, I,
↪  unsafe_mat, unsafe_vec)
print('Number of Over-approximate Reachable sets: ',
↪  len(R_approx))
print('Number of Over-approximate Unsafe sets: ', len(U_approx))
print('Verification result: ', Res_approx)
plot_star(R_approx)
plot_star(U_approx)
```

```
-------------------------------------------
Exact Qualitative Verification:
-------------------------------------------
Computing layer 0 FullyConnectedLayer reachable set...
Number of stars/probstars: 1
Computing layer 1 ReLULayer reachable set...
Number of stars/probstars: 6
Computing layer 2 FullyConnectedLayer reachable set...
Number of stars/probstars: 6

Number of Exact Reachable sets:  6
Number of Exact Unsafe sets:  6
Number of Counter Input sets:  6
Verification result:  ['SAT', 'SAT', 'SAT', 'SAT', 'SAT', 'SAT']

-------------------------------------------
Over-approximate Qualitative Verification:
-------------------------------------------
Computing layer 0 FullyConnectedLayer reachable set...
Number of stars: 1
Computing layer 1 ReLULayer reachable set...
Number of stars: 1
Computing layer 2 FullyConnectedLayer reachable set...
Number of stars: 1

Number of Over-approximate Reachable sets:  1
Number of Over-approximate Unsafe sets:  1
Verification result:  UNKNOWN
```

1.5 Quantitative Reachability Analysis of FFNN

To perform quantitative reachability of a network, StarV computes the reachable sets of the network layer-by-layer with input defined as ProbStar set. Additionally, StarV supports both exact and over-approximate reachability methods



(a) `plot_star(U_exact).` (b) `plot_star(C_exact).` (c) `plot_star(U_approx).`

Figure 5.2: Visualization of Verification results of an FFNN with Star.

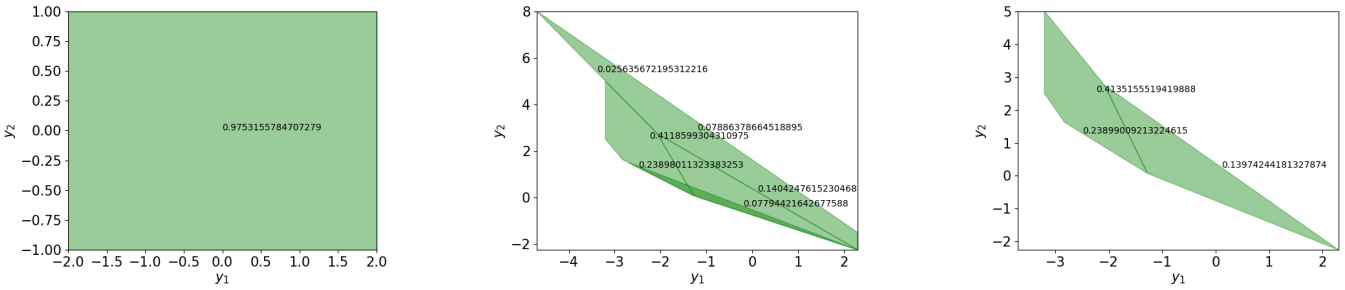
for the network with piecewise linear activation functions. The exact scheme is similar to the qualitative reachability method. For an over-approximate scheme, we compute and filter out reachable sets whose probability is lower than a user-defined threshold for each layer. Therefore, users can use multiple cores to speed up the reachability analysis for both schemes for the quantitative reachability of a network.

Definition 8 (Probabilistic Reachability of FFNN)**.** *Given a Feedforward Neural Networks network $\mathcal{F}$ and a constrained probabilistic input set, $X = \{x \in \mathbb{R}^n | \; Cx \leq d \wedge x \sim \mathcal{N}(\mu, \Sigma)\}$, the probabilistic reachability analysis of the network $\mathcal{F}$ is the process of computing the probabilistic output set of the network and its associate probability, i.e., $Y = \mathcal{F}(x), x \in X$.*

The following example shows how to conduct exact and over-approximate reachability analysis for the FFNN using the ProbStar set. We can see from the results that 3 reachable sets are filtered out from the over-approximate reachability analysis, as shown in Figure 5.3c.

```
#  ... Construct a feedforward neural network ...

# Construct input set
lb = np.array([-2.0, -1.0])
ub = np.array([2.0, 1.0])
S = Star(lb, ub)
mu = 0.5*(S.pred_lb + S.pred_ub)
a = 2.5 # coefficience to adjust the distribution
sig = (mu - S.pred_lb)/a
Sig = np.diag(np.square(sig))
P = ProbStar(S.V, S.C, S.d, mu, Sig, S.pred_lb, S.pred_ub)
plot_probstar(P)

# ----- Parallelized version -----
numCores = 2
pool = multiprocessing.Pool(numCores)
# ----- Single-core version -----
# pool = None

# Exact Reachability analysis
I = [P]
R_exact = reachExactBFS(F, I, lp_solver='gurobi', pool=pool,
↪  show=False)
print('Number of Exact Reachable sets: ', len(R_exact))
plot_probstar(R_exact)

# Over-approximation reachability analysis
I = [P]
R_approx, p_ignored = reachApproxBFS(F, I, p_filter=0.1,
↪  lp_solver='gurobi', pool=pool, show=True)
print('Number of Over-approximate Reachable sets: ',
↪  len(R_approx))
plot_probstar(R_approx)
```

```
-----------------------------------------------
Exact Reachability Analysis:
-----------------------------------------------
Number of Exact Reachable sets:  6

-----------------------------------------------
Over-approximate Reachability Analysis:
-----------------------------------------------
================ Layer 0 =================
Computing layer 0 reachable set...
Number of probstars: 1
Filtering probstars whose probabilities < 0.1...
Number of ignored probstars: 0
Number of remaining probstars: 1
================ Layer 1 =================
Computing layer 1 reachable set...
Number of probstars: 6
Filtering probstars whose probabilities < 0.1...
Number of ignored probstars: 3
Number of remaining probstars: 3
================ Layer 2 =================
Computing layer 2 reachable set...
Number of probstars: 3
Filtering probstars whose probabilities < 0.1...
Number of ignored probstars: 0
Number of remaining probstars: 3

Number of Over-approximate Reachable sets:  3
```



(a) `plot_probstar(P)`. (b) `plot_probstar(R_exact)`. (c) `plot_probstar(R_approx)`.

Figure 5.3: Visualization of Reachability Analysis on an FFNN with ProbStar.

1.6 Quantitative Verification of FFNN

Similar to the qualitative verification of FFNN, users can conduct quantitative verification of FFNN manually or automatically after performing a reachability analysis of the network. The quantitative verification of FFNN is formally defined in Def. 9.

Definition 9 (Quantitative Verification of FFNN)**.** *Given a Feedforward Neural Networks network $\mathcal{F}$, a constrained probabilistic input set $X = \{x \in \mathbb{R}^n | \ Cx \leq d \ \wedge \ x \sim \mathcal{N}(\mu, \Sigma)\}$, and a linear safety specification $S \triangleq Hy \leq g$, where y is the output of the network, the quantitative verification of the network is the process of computing the probability of the network satisfying its specification, i.e., $\mathcal{P}(Hy \leq g, y \in Y = \mathcal{F}(x), x \in X)$, where $\mathcal{P}$ states for a probability.*

In this example, we illustrate how to conduct quantitative verification of the FFNN automatically using StarV. Assume we want to verify the following property:

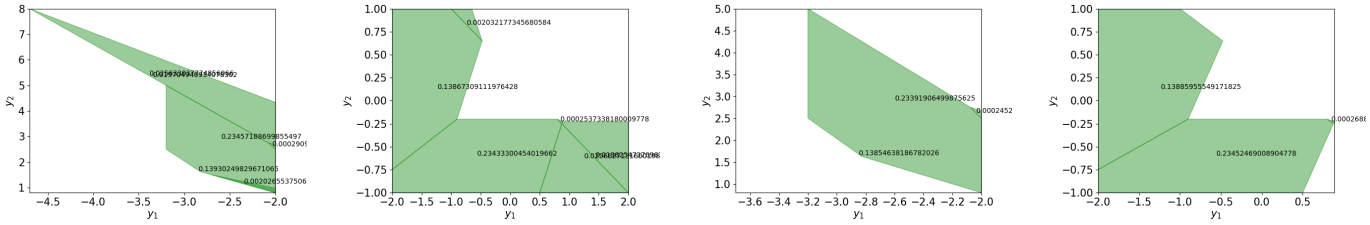
$$S = \{y \in \mathbb{R}^2 | y_1 \leq -2\} \tag{5.3}$$

```
# ... Construct a feedforward neural network ...

# ... Construct input set ...

# Specifying a property of an FFNN
# y1 <= -2
unsafe_mat = np.array([[1.0, 0]])
unsafe_vec = np.array([-2.0])

# number of cores
numCores = 2

I = [P]

# Exact Verification
R_exact, U_exact, C_exact, _, _, _, _ = quantiVerifyBFS(F, I,
    unsafe_mat, unsafe_vec, p_filter=0.0, numCores=numCores,
    show=False)
print('Number of Exact Reachable sets: ', len(R_exact))
print('Number of Exact Unsafe sets: ', len(U_exact))
print('Number of Counter Input sets: ', len(C_exact))
plot_probstar(R_exact)
plot_probstar(U_exact)
plot_probstar(C_exact)

# Over-approximate Verification
R_approx, U_approx, C_approx, _, _, _, _ = quantiVerifyBFS(F, I,
    unsafe_mat, unsafe_vec, p_filter=0.1, numCores=numCores,
    show=True)
print('Number of Over-approximate Reachable sets: ',
    len(R_approx))
print('Number of Over-approximate Unsafe sets: ', len(U_approx))
print('Number of Counter Input sets: ', len(C_approx))
plot_probstar(R_approx)
plot_probstar(U_approx)
```

```
-------------------------------------------
Exact Qualitative Verification:
-------------------------------------------
Number of Exact Reachable sets:  6
Number of Exact Unsafe sets:  6
Number of Counter Input sets:  6

-------------------------------------------
Over-approximate Qualitative Verification:
-------------------------------------------
================ Layer 0 =================
Computing layer 0 reachable set...
Number of probstars: 1
Filtering probstars whose probabilities < 0.1...
Number of ignored probstars: 0
Number of remaining probstars: 1
================ Layer 1 =================
Computing layer 1 reachable set...
Number of probstars: 6
Filtering probstars whose probabilities < 0.1...
Number of ignored probstars: 3
Number of remaining probstars: 3
================ Layer 2 =================
Computing layer 2 reachable set...
Number of probstars: 3
Filtering probstars whose probabilities < 0.1...
Number of ignored probstars: 0
Number of remaining probstars: 3

Number of Over-approximate Reachable sets:  3
Number of Over-approximate Unsafe sets:  3
Number of Counter Input sets:  3
```

```
plot_probstar(C_approx)
```

From the results, we can see that both exact and over-approximate methods can prove the network is unsafe. For the exact scheme, all of the reachable sets of the network reach the unsafe region, and we identify the counterexample corresponding to each unsafe output set. For the over-approximate scheme, we still need to perform exact reachability analysis for each layer and then filter out partial reachable sets whose probabilities are lower than the predefined threshold. We can still trace back to find the exact counterexample if any of the remaining output reachable sets reach the unsafe region. The visualization of the results is shown in Figure 5.4.



(a) `plot_probstar(U_exact).` (b) `plot_probstar(C_exact).` (c) `plot_probstar(U_approx).` (d) `plot_probstar(U_approx).`

Figure 5.4: Visualization of Verification results on an FFNN with ProbStar.

2 Verification of Convolutional Neural Networks (CNN)

2.1 Qualitative Reachability Analysis of CNN

Definition 10 (Reachability of a CNN). *Given a bounded convex polytope input set $\mathcal{I} \triangleq \{x \mid Ax \leq b\}$, the reachability analysis of a CNN $\mathcal{N} = \{L_i\}$, where $i \in [1, 2, \cdots, l]$, is analyzed layer-by-layer computing an output reachable set $\mathcal{R}_\mathcal{N}$ as follows,*

$$\begin{aligned}
\mathcal{R}_{L_1} &\triangleq \{y_1 \mid y_1 = L_1(x), x \in \mathcal{I}\},\\
\mathcal{R}_{L_2} &\triangleq \{y_2 \mid y_2 = L_2(y_1), y_1 \in \mathcal{R}_{L_1}\},\\
&\vdots\\
\mathcal{R}_\mathcal{N} = \mathcal{R}_{L_l} &\triangleq \{y_l \mid y_l = L_l(y_{l-1}), y_{l-1} \in \mathcal{R}_{L_{l-1}}\},
\end{aligned}$$

where $L_i \in \{L_{conv}, L_{avg}, L_{max}, L_{bn}, L_r, L_{flat}, L_{fc}\}$ is the i^{th} layer operation.

Reachability analysis of CNN using ImageStar.

```
dtype='float32'
net_type = 'Small'
folder_dir = f'{path}/util/data/nets/CAV2020_MNIST_ConvNet'
net_dir = f"{folder_dir}/onnx/{net_type}_ConvNet.onnx"
print(net_dir)
starvNet = load_convnet(net_dir, net_type, dtype=dtype)
print()
print(starvNet.info())

mat_file =
↪  scipy.io.loadmat(f"{folder_dir}/nnv/{net_type}_images.mat")
data = mat_file['IM_data'].astype(dtype)[:, :, 0]
label = (mat_file['IM_labels'] - 1)[0]

delta = 0.005
d = 250
lb, ub = brightening_attack(data, delta=delta, d=d, dtype=dtype)
IM = ImageStar(lb, ub)
print('Input ImageStar:')
repr(IM)

```

```
# print(StarVNet.info())
==============NETWORK===============
Network type: smallConvNetMNIST_CAV2020
Input Dimension: 1
Output Dimension: 7
Number of Layers: 14
Layer types:
Layer 0: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'>
(1, 1, dtype=float32)
Layer 1: <class 'StarV.layer.Conv2DLayer.Conv2DLayer'>
(1, 8, kernel_size = (3, 3), stride = [1 1], padding = [1 1 1 1],
↪  dtype=float32)
Layer 2: <class 'StarV.layer.BatchNorm2DLayer.BatchNorm2DLayer'>
(8, eps=9.999999747378752e-06, dtype=float32)
Layer 3: <class 'StarV.layer.ReLULayer.ReLULayer'>
Layer 4: <class 'StarV.layer.MaxPool2DLayer.MaxPool2DLayer'>
(kernel_size = [2 2], stride = [2 2], padding = [0 0])
Layer 5: <class 'StarV.layer.Conv2DLayer.Conv2DLayer'>
(8, 16, kernel_size = (3, 3), stride = [1 1], padding = [1 1 1 1],
↪  dtype=float32)
Layer 6: <class 'StarV.layer.BatchNorm2DLayer.BatchNorm2DLayer'>
(16, eps=9.999999747378752e-06, dtype=float32)
Layer 7: <class 'StarV.layer.ReLULayer.ReLULayer'>
Layer 8: <class 'StarV.layer.MaxPool2DLayer.MaxPool2DLayer'>
(kernel_size = [2 2], stride = [2 2], padding = [0 0])
Layer 9: <class 'StarV.layer.Conv2DLayer.Conv2DLayer'>
```

```
reachMethod = 'approx'
lp_solver = 'gurobi'
pool = None
RF = 0.0
DR = 0
show  = False
R, _ = reachBFS(net=starvNet, inputSet=IM,
↪  reachMethod=reachMethod, lp_solver=lp_solver, pool=pool,
↪  RF=RF, DR=DR, show=show)
print('Output ImageStar after reachability analysis:')
repr(R)
```

```
(16, 32, kernel_size = (3, 3), stride = [1 1], padding = [1 1 1 1],
↪  dtype=float32)
Layer 10: <class 'StarV.layer.BatchNorm2DLayer.BatchNorm2DLayer'>
(32, eps=9.999999747378752e-06, dtype=float32)
Layer 11: <class 'StarV.layer.ReLULayer.ReLULayer'>
Layer 12: <class 'StarV.layer.Conv2DLayer.Conv2DLayer'>
(32, 10, kernel_size = (7, 7), stride = [1 1], padding = [0 0 0 0],
↪  dtype=float32)
Layer 13: <class 'StarV.layer.FlattenLayer.FlattenLayer'>
(channel_last=True)

Input ImageStar:
ImageStar Set:
V: (28, 28, 1, 10), float32
C: (0, 0), float32
d: (0,), float32
pred_lb: (9,), float32
pred_ub: (9,), float32
height: 28
width: 28
num_channel: 1
num_pred: 9

Output ImageStar after reachability analysis:
ImageStar Set:
V: (1, 1, 10, 15), float32
C: (15, 14), float32
d: (15,), float32
pred_lb: (14,), float32
pred_ub: (14,), float32
height: 1
width: 1
num_channel: 10
num_pred: 14
```

Reachability analysis of CNN using SparseImageStar COO.

```
mat_file =
↪  scipy.io.loadmat(f"{folder_dir}/nnv/{net_type}_images.mat")
data = mat_file['IM_data'].astype(dtype)[:, :, 0]
label = (mat_file['IM_labels'] - 1)[0]

delta = 0.005
d = 250
lb, ub = brightening_attack(data, delta=delta, d=d, dtype=dtype)
COO = SparseImageStar2DCOO(lb, ub)
print('Input SparseImageStar COO:')
repr(COO)

reachMethod = 'approx'
lp_solver = 'gurobi'
pool = None
RF = 0.0
DR = 0
show  = False
R, _ = reachBFS(net=starvNet, inputSet=COO,
↪  reachMethod=reachMethod, lp_solver=lp_solver, pool=pool,
↪  RF=RF, DR=DR, show=show)
print('Output SparseImageStar COO after reachability analysis:')
repr(R)
```

```
Input SparseImageStar COO:
SparseImageStar2DCOO Set:
c: (784,), float32
V_coo: (784, 9), data: float32, row: int32, col: int32
C_csr: (0, 0), float32
d: (0,), float32
pred_lb: (9,), float32
pred_ub: (9,), float32
shape: (28, 28, 1)
num_pred: 9
density: 0.0012755102040816326
nnz: 9

Output SparseImageStar COO after reachability analysis:
SparseImageStar2DCOO Set:
c: (10,), float32
V_coo: (10, 14), data: float32, row: int32, col: int32
C_csr: (15, 14), float32
d: (15,), float32
pred_lb: (14,), float32
pred_ub: (14,), float32
shape: (10,)
num_pred: 14
density: 0.8571428571428571
nnz: 120
```

Reachability analysis of CNN using SparseImageStar CSR.

```
mat_file =
↪  scipy.io.loadmat(f"{folder_dir}/nnv/{net_type}_images.mat")
data = mat_file['IM_data'].astype(dtype)[:, :, 0]
label = (mat_file['IM_labels'] - 1)[0]

delta = 0.005
d = 250
lb, ub = brightening_attack(data, delta=delta, d=d, dtype=dtype)
CSR = SparseImageStar2DCSR(lb, ub)
print('Input SparseImageStar CSR:')
repr(CSR)

reachMethod = 'approx'
```

```
Input SparseImageStar CSR:
SparseImageStar2DCSR Set:
c: (784,), float32
V_csr: (784, 9), data: float32, indices: int32, indptr: int32
C_csr: (0, 0), float32
d: (0,), float32
pred_lb: (9,), float32
pred_ub: (9,), float32
shape: (28, 28, 1)
num_pred: 9
density: 0.0012755102040816326
nnz: 9

Output SparseImageStar CSR after reachability analysis:
SparseImageStar2DCSR Set:
c: (10,), float32
V_csr: (10, 14), data: float32, indices: int32, indptr: int32
```

```
lp_solver = 'gurobi'
pool = None
RF = 0.0
DR = 0
show  = False
R, _ = reachBFS(net=starvNet, inputSet=CSR, reachMethod=reachMethod, lp_solver=lp_solver, pool=pool, RF=RF, DR=DR, show=show)
print('Output SparseImageStar CSR after reachability analysis:')
repr(R)
print(R)
```

```
C_csr: (15, 14), float32
d: (15,), float32
pred_lb: (14,), float32
pred_ub: (14,), float32
shape: (10,)
num_pred: 14
density: 0.8571428571428571
nnz: 120
```

2.2 Certifying the robustness of CNN

Certifying CNN using ImageStar.

```
print('Input ImageStar:')
repr(IM)

reachMethod = 'approx'
lp_solver = 'gurobi'
pool = None
RF = 0.0
DR = 0
show  = False
veriMethod = 'BFS'
rb, vt, _, _ = certifyRobustness(net=starvNet, inputs=IM, labels=label,
            veriMethod=veriMethod, reachMethod=reachMethod, lp_solver=lp_solver, pool=pool,
            RF=RF, DR=DR, return_output=False, show=show)
rb = 'Robust' if rb else 'Unknown'
print('Certification result: ', rb)
print(f'Verification time: {vt} sec')
```

```
Input ImageStar:
ImageStar Set:
V: (28, 28, 1, 10), float32
C: (0, 0), float32
d: (0,), float32
pred_lb: (9,), float32
pred_ub: (9,), float32
height: 28
width: 28
num_channel: 1
num_pred: 9

Certification result:  Robust
Verification time: 0.2618976879748516 sec
```

Certifying CNN using SparseImageStar COO.

```
print('Input SparseImageStar COO:')
repr(COO)

reachMethod = 'approx'
lp_solver = 'gurobi'
pool = None
RF = 0.0
DR = 0
show  = False
veriMethod = 'BFS'
rb, vt, _, _ = certifyRobustness(net=starvNet, inputs=COO, labels=label,
            veriMethod=veriMethod, reachMethod=reachMethod, lp_solver=lp_solver, pool=pool,
            RF=RF, DR=DR, return_output=False, show=show)
rb = 'Robust' if rb else 'Unknown'
print('Certification result: ', rb)
print(f'Verification time: {vt} sec')
```

```
Input SparseImageStar COO:
SparseImageStar2DCOO Set:
c: (784,), float32
V_coo: (784, 9), data: float32, row: int32, col: int32
C_csr: (0, 0), float32
d: (0,), float32
pred_lb: (9,), float32
pred_ub: (9,), float32
shape: (28, 28, 1)
num_pred: 9
density: 0.0012755102040816326
nnz: 9

Certification result:  Robust
Verification time: 0.18907868000678718 sec
```

Certifying CNN using SparseImageStar CSR.

```
print('Input SparseImageStar CSR:')
repr(CSR)

reachMethod = 'approx'
lp_solver = 'gurobi'
pool = None
RF = 0.0
DR = 0
```

```
Input SparseImageStar CSR:
SparseImageStar2DCSR Set:
c: (784,), float32
V_csr: (784, 9), data: float32, indices: int32, indptr: int32
C_csr: (0, 0), float32
d: (0,), float32
pred_lb: (9,), float32
pred_ub: (9,), float32
shape: (28, 28, 1)
num_pred: 9
density: 0.0012755102040816326
```

```
show  = False
veriMethod = 'BFS'
rb, vt, _, _ = certifyRobustness(net=starvNet, inputs=CSR, labels=label,
            veriMethod=veriMethod, reachMethod=reachMethod, lp_solver=lp_solver, pool=pool,
            RF=RF, DR=DR, return_output=False, show=show)
rb = 'Robust' if rb else 'Unknown'
print('Certification result: ', rb)
print(f'Verification time: {vt} sec')
```

```
nnz: 9

Certification result:  Robust
Verification time: 0.24403263899148442 sec
```

3 Verification of Recurrent Neural Networks (RNN)

3.1 Long short-term memory (LSTM) RNN

Qualitative Reachability Analysis of LSTM RNN using SparseStar

```
type='lstm'
in_shape = 784
num_seq = 2
frame_size = in_shape // num_seq
hidden=15

# load dataset
data_dir = f'{path}/util/data/nets/NAHS2024_RNN'
with open(f'{data_dir}/mnist_test.csv', 'r') as x:
    test_data = list(csv.reader(x, delimiter=","))

test_data = np.array(test_data)
# data
XTest = copy.deepcopy(test_data[:, 1:]).astype('float32').T / 255
XTest = XTest.reshape([frame_size, num_seq, 10000], order='F')
data = XTest[:, :, 0]

# load LSTM neural network
net_name = f'MNIST_{type.upper()}{hidden}net'
net_dir = f'{data_dir}/{net_name}.onnx'
net = load_LSTM_network(net_dir, net_name)

epsilon = 0.005
# Construct input sequential SparseStar sets
print('Input sequential SparseStar sets:')
X = []
for i in range(num_seq):
    S = SparseStar(np.clip(data[:, i] - epsilon, 0, 1),
                   np.clip(data[:, i] + epsilon, 0, 1))
    X.append(S)
    print(f'sequence {i}: \n')
    repr(S)
    print()

# Reachability Analysis
Y, _ = reachBFS(net=net, inputSet=X, reachMethod='approx', lp_solver='gurobi')
print('Output SparseStar after reachability analysis:')
print(Y)
```

```
Input sequential SparseStar sets:

sequence 0:
SparseStar Set:
A: (392, 393)
C_csc: (0, 392)
d: (0,)
pred_lb: (392,)
pred_ub: (392,)
pred_depth: (392,)
dim: 392
nVars: 392
nZVars: 0
nIVars: 392

sequence 1:
SparseStar Set:
A: (392, 393)
C_csc: (0, 392)
d: (0,)
pred_lb: (392,)
pred_ub: (392,)
pred_depth: (392,)
dim: 392
nVars: 392
nZVars: 0
nIVars: 392

Output SparseStar after reachability analysis:
SparseStar Set:
A: (10, 16)
C_csc: (360, 7236)
d: (360,)
pred_lb: (7236,)
pred_ub: (7236,)
pred_depth: (7236,)
dim: 10
nVars: 7236
nZVars: 7221
nIVars: 15
```

Certifying the robustness LSTM RNN using SparseStar

```
type='lstm'
in_shape = 784
num_seq = 2
frame_size = in_shape // num_seq
hidden=15

# load dataset
data_dir = f'{path}/util/data/nets/NAHS2024_RNN'
```

```
Output SparseStar after reachability analysis:
SparseStar Set:
A: (10, 16)
C_csc: (360, 7236)
d: (360,)
pred_lb: (7236,)
pred_ub: (7236,)
pred_depth: (7236,)
dim: 10
nVars: 7236
nZVars: 7221
```

```
with open(f'{data_dir}/mnist_test.csv', 'r') as x:
    test_data = list(csv.reader(x, delimiter=","))

test_data = np.array(test_data)
# data
XTest = copy.deepcopy(test_data[:, 1:]).astype('float32').T / 255
XTest = XTest.reshape([frame_size, num_seq, 10000], order='F')
data = XTest[:, :, 0]

# load LSTM neural network
net_name = f'MNIST_{type.upper()}{hidden}net'
net_dir = f'{data_dir}/{net_name}.onnx'
net = load_LSTM_network(net_dir, net_name)

Y, rb, vt = certifyRobustness_sequence(net, data, epsilon=0.005,
↪  lp_solver='gurobi', DR=0, show=False)
print('Output SparseStar after reachability analysis:')
repr(Y)
rb = 'Robust' if rb[0] else 'Unknown'
print('Certification result: ', rb)
print(f'Verification time: {vt} sec')
```

```
nIVars: 15

Certification result:  Robust
Verification time: 9.7772582769976 sec
```

3.2 Gated Recurrent Unit (GRU) RNN

Qualitative Reachability Analysis of GRU RNN using SparseStar

```
type='gru'
in_shape = 784
num_seq = 2
frame_size = in_shape // num_seq
hidden=15

# load dataset
data_dir = f'{path}/util/data/nets/NAHS2024_RNN'
with open(f'{data_dir}/mnist_test.csv', 'r') as x:
    test_data = list(csv.reader(x, delimiter=","))

test_data = np.array(test_data)
# data
XTest = copy.deepcopy(test_data[:, 1:]).astype('float32').T / 255
XTest = XTest.reshape([frame_size, num_seq, 10000], order='F')
data = XTest[:, :, 0]

# load GRU neural network
net_name = f'MNIST_{type.upper()}{hidden}net'
net_dir = f'{data_dir}/{net_name}.onnx'
net = load_GRU_network(net_dir, net_name)

epsilon = 0.005
# Construct input sequential SparseStar sets
print('Input sequential SparseStar sets:')
X = []
for i in range(num_seq):
    S = SparseStar(np.clip(data[:, i] - epsilon, 0, 1),
                   np.clip(data[:, i] + epsilon, 0, 1))
    X.append(S)
    print(f'sequence {i}: \n')
    repr(S)
    print()

# Reachability Analysis
Y, _ = reachBFS(net=net, inputSet=X, reachMethod='approx',
↪  lp_solver='gurobi')
print('Output SparseStar after reachability analysis:')
repr(Y)
```

```
Input sequential SparseStar sets:
sequence 0:
SparseStar Set:
A: (392, 393)
C_csc: (0, 392)
d: (0,)
pred_lb: (392,)
pred_ub: (392,)
pred_depth: (392,)
dim: 392
nVars: 392
nZVars: 0
nIVars: 392

sequence 1:
SparseStar Set:
A: (392, 393)
C_csc: (0, 392)
d: (0,)
pred_lb: (392,)
pred_ub: (392,)
pred_depth: (392,)
dim: 392
nVars: 392
nZVars: 0
nIVars: 392

Output SparseStar after reachability analysis:
SparseStar Set:
A: (10, 31)
C_csc: (540, 7643)
d: (540,)
pred_lb: (7643,)
pred_ub: (7643,)
pred_depth: (7643,)
dim: 10
nVars: 7643
nZVars: 7613
nIVars: 30
```

Certifying the robustness GRU RNN using SparseStar

```
type='gru'
in_shape = 784
num_seq = 2
frame_size = in_shape // num_seq
hidden=15

# load dataset
data_dir = f'{path}/util/data/nets/NAHS2024_RNN'
with open(f'{data_dir}/mnist_test.csv', 'r') as x:
    test_data = list(csv.reader(x, delimiter=","))

test_data = np.array(test_data)
# data
XTest = copy.deepcopy(test_data[:, 1:]).astype('float32').T / 255
XTest = XTest.reshape([frame_size, num_seq, 10000], order='F')
data = XTest[:, :, 0]

# load GRU neural network
net_name = f'MNIST_{type.upper()}{hidden}net'
net_dir = f'{data_dir}/{net_name}.onnx'
net = load_GRU_network(net_dir, net_name)

Y, rb, vt = certifyRobustness_sequence(net, data, epsilon=0.005,
↪  lp_solver='gurobi', DR=0, show=False)
print('Output SparseStar after reachability analysis:')
repr(Y)
rb = 'Robust' if rb[0] else 'Unknown'
print('Certification result: ', rb)
print(f'Verification time: {vt} sec')
```

```
Output SparseStar after reachability analysis:
SparseStar Set:
A: (10, 31)
C_csc: (540, 7643)
d: (540,)
pred_lb: (7643,)
pred_ub: (7643,)
pred_depth: (7643,)
dim: 10
nVars: 7643
nZVars: 7613
nIVars: 30

Certification result:  Robust
Verification time: 14.583610269997735 sec
```

Chapter 6

Verification of Neural Network Control Systems(NNCS)

Problem 1 (Qualitative Verification of NNCS)**.** *Given a CPS with an FFNN controller F, a discrete, and linear plant P with the initial $x(0)$ in an initial set X_0, verify whether or not the state of the plant satisfies a safety property in a bounded time steps k_{max}. Formally, we want to verify if $\forall x(0) \in X_0 \rightarrow g(x(k)) \models S(g(x(k))), \forall 0 \leq k \leq k_{max}$ in which g is a nonlinear transformation function and S is a linear predicate over the transformed state variables $g(x(k))$ defining the safety requirements of the system.*

Definition 11 (Quantitative verification of Le-CPS)**.** *Given a Le-CPS system with a ReLU FFNN controller F, and a discrete linear plant $\mathcal{M}$ with the initial state $x(0)$ in an initial constrained probabilistic input set $X_0 = \{x(0) \in \mathbb{R}^n | \ Cx(0) \leq d \ \wedge \ x(0) \sim \mathcal{N}(\mu, \Sigma)\}$, the quantitative verification involves verifying whether or not the state of the plant satisfies a safety property in a bounded time steps k_{max} and computing the satisfaction probability. Formally, we want to verify if $\forall x(0) \in X_0 \rightarrow x(k) \models S(x(k)), 0 \leq k \leq k_{max}$ and compute the probabilities P_i of the satisfaction at each time step $\mathcal{P} = \{P_0, P_1, \ldots, P_{k_{max}}\}, P_i = P(x(i) \models S(x(i))$ in which S is a linear predicate over the state variables $x(k)$ defining the safety requirements of the system.*

Definition 12 (Counterexample Construction for Le-CPS)**.** *Given a Learning-enabled CPS as described in Definition 11 with an initial state $x(0) \in X_0$, the counterexample construction determines the complete ProbStar set of initial conditions that lead the system to violate the safety property S at any step k within the bounded time horizon k_{max}.*

1 NNCS architecture

StarV supports the qualitative and quantitative verification of closed-loop control systems with an **FFNN controller with piecewise linear activation functions** using ProbStar. The architecture of such a system is depicted in Figure 6.1. The plant model in the system can be continuous, discrete, and linear.

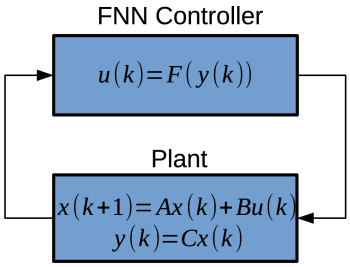


Figure 6.1: Neural Network Control System (NNCS)

2 Main Steps

Using StarV for the verification of a neural network control system (NNCS) consists of the following main steps:

- Constructing an NNCS object.

- Constructing an initial set of states of the system.
- Specifying a property of the system to verify.
- Choosing a reachability analysis method.
- Verifying the system.

In this section, we will use the learning-based Adaptive Cruise Control (ACC) system [11] as an example to illustrate how to conduct verification of a NNCS using StarV. To start, we first introduce the ACC system briefly.

3 Example: Verification of ACC System

The learning-based Adaptive Cruise Control (ACC) system [11] comprises an ego (following) vehicle and a lead vehicle. The ego vehicle is equipped with a radar sensor that measures the distance to the lead vehicle in the same lane, denoted as D_r, and the relative velocity of the lead vehicle, denoted as V_r. In Speed Control Mode, the ego vehicle maintains a user-set speed of $V_{set} = 30\,\mathrm{m/s}$. In Spacing Control Mode, the ego vehicle is required to maintain a safe distance from the lead vehicle, defined by D_{safe}. Two neural networks with N layers ($N = 3, 5$) and 20 neurons per layer utilizing ReLU activation functions, are trained to control the ego vehicle with a control period of 0.1 seconds. In this example, we are interested in the network $N_{5\times 20}$. The system's linear dynamics are modeled as follows:

$$
\begin{aligned}
&\dot{x}_{lead}(t) = v_{lead}(t),\ \dot{v}_{lead}(t) = \gamma_{lead},\\
&\dot{\gamma}_{lead}(t) = -2\gamma_{lead}(t) + 2a_{lead},\\
&\dot{x}_{ego}(t) = v_{ego}(t),\ \dot{v}_{ego}(t) = \gamma_{ego},\\
&\dot{\gamma}_{ego}(t) = -2\gamma_{ego}(t) + 2a_{ego},
\end{aligned}
$$

where $x_{lead}(x_{ego})$, $v_{lead}(v_{ego})$ and $\gamma_{lead}(\gamma_{ego})$ are the position, velocity, and acceleration of the lead (ego) vehicle respectively. $a_{lead}(a_{ego})$ is the acceleration control input applied to the lead (ego) vehicle. To obtain a discrete linear plant model, the continuous-time dynamics are discretized using a zero-order hold on the inputs with a sampling time of 0.1 seconds, corresponding to the control period.

Safety Scenarios. In the Speed Control Mode, when the ego vehicle maintains a safe distance, the lead vehicle driver may suddenly decelerate with $a_{lead} = -5\,\mathrm{m/s^2}$ to reduce speed. The neural network controller of the ego vehicle is required to decelerate appropriately to maintain a safe distance D_r from the lead vehicle within a maximum of 5 seconds. The ***safety property*** is defined as:

$$D_r = x_{lead} - x_{ego} \geq D_{safe} = D_{default} + T_{gap} \times v_{ego}, \tag{6.1}$$

where $T_{gap} = 1.4\,\text{seconds}$ and $D_{default} = 10\,\text{meters}$. The system's safety is evaluated under the following ***initial conditions***: $x_{lead}(0) \in [90, 92]\,\text{meters}$, $v_{lead}(0) \in [20, 21]\,\mathrm{m/s}$, $\gamma_{lead}(0) = \gamma_{ego}(0) = 0\,\mathrm{m/s^2}$, $v_{ego}(0) \in [30, 30.5]\,\mathrm{m/s}$, $x_{ego}(0) \in [30, 31]\,\text{meters}$.

3.1 Constructing the ACC system

An NNCS object is constructed using an FFNN controller object and a continuous/discrete linear plant model object. In this example, we constructed the `controller_5_20` using loaded weights and biases, and the ACC system using its system dynamics.

```
# Load the neural network controller
starv_root_path = os.path.dirname(StarV.__file__)
net_path = starv_root_path +
↪  '/util/data/nets/ACC/controller_5_20.mat'
mat_contents = loadmat(net_path)
W = mat_contents['W']
b = mat_contents['b']

n = W.shape[1]
layers = []
for i in range(0,n-1):
    Wi = W[0,i]
    bi = b[i,0]
    bi = bi.reshape(bi.shape[0],)
```

```
================NEURAL NETWORK CONTROL SYSTEM================
nncs-type: DLNNCS
number of outputs: 3
number of inputs: 5
number of feeback inputs: 3
numher of reference inputs: 2
network controller & plant model information:

=============NETWORK===============
Network type: controller_5_20
Input Dimension: 5
Output Dimension: 1
Number of Layers: 11
Layer types:
Layer 0: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (20, 5,
↪  dtype=float64)
Layer 1: <class 'StarV.layer.ReLULayer.ReLULayer'>
```

```
    L1 = FullyConnectedLayer([Wi, bi])
    L2 = ReLULayer()
    layers.append(L1)
    layers.append(L2)

bi = b[n-1,0]
bi = bi.reshape(bi.shape[0],)
L1 = FullyConnectedLayer([W[0,n-1], bi])
layers.append(L1)

net = NeuralNetwork(layers, 'controller_5_20')

# Load the plant dynamics
A = np.array([[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., 0., 0., 0., 0., -2.]])
B = np.array([[0.], [0.], [0.], [0.], [0.], [2.], [0.]])
C = np.array([[1., 0., 0., -1., 0., 0., 0.],
                [0., 1., 0., 0., -1., 0., 0.],
                [0., 0., 0., 0., 1., 0., 0.]])
# feedbacks:
# 1) relative distance: x1 - x4
# 2) relative velocity: x2 - x5
# 3) longtitudinal velocity: x5

D = np.array([[0.], [0.], [0.]])

plant_model = LODE(A, B, C, D)
dplant = plant_model.toDLODE(0.1)  # dt = 0.1

sys = NNCS(net, dplant, type='DLNNCS')

print(sys)
```

```
Layer 2: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (20, 20,
↪  dtype=float64)
Layer 3: <class 'StarV.layer.ReLULayer.ReLULayer'>
Layer 4: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (20, 20,
↪  dtype=float64)
Layer 5: <class 'StarV.layer.ReLULayer.ReLULayer'>
Layer 6: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (20, 20,
↪  dtype=float64)
Layer 7: <class 'StarV.layer.ReLULayer.ReLULayer'>
Layer 8: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (20, 20,
↪  dtype=float64)
Layer 9: <class 'StarV.layer.ReLULayer.ReLULayer'>
Layer 10: <class
↪  'StarV.layer.FullyConnectedLayer.FullyConnectedLayer'> (1, 20,
↪  dtype=float64)

========= DLODE ==========
Plant Matrices:

A = [[1.      0.1     0.005   0.      0.      0.      0.00016]
[0.      1.      0.1     0.      0.      0.      0.00468]
[0.      0.      1.      0.      0.      0.      0.09063]
[0.      0.      0.      1.      0.1     0.00468 0.     ]
[0.      0.      0.      0.      1.      0.09063 0.     ]
[0.      0.      0.      0.      0.      0.81873 0.     ]
[0.      0.      0.      0.      0.      0.      0.81873]]

B = [[0.     ]
[0.     ]
[0.     ]
[0.00032]
[0.00937]
[0.18127]
[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.]]

Number of inputs: 1
Number of outputs: 3
```

3.2 Constructing the Initial Set of States

To perform quantitative verification of the ACC system, we need to define a probabilistic initial set of states using ProbStars.

```
# input sets (multiple input set - 6 individual depending on v_lead_0)
x_lead_0 = [90., 92.]
v_lead_0 = [[29., 30.], [28., 29.], [27., 28.], [26., 27.], [25., 26.], [20., 21.]]
acc_lead_0 = [0., 0.]
x_ego_0 = [30., 31.,]
v_ego_0 = [30., 30.5]
acc_ego_0 = [0., 0.]
a_lead = -5.0
x7_0 = [2*a_lead, 2*a_lead]

initSets = []
for i in range(0, 6):
    v_lead_0_i = v_lead_0[i]
    lb = np.array([x_lead_0[0], v_lead_0_i[0], acc_lead_0[0], x_ego_0[0], v_ego_0[0], acc_ego_0[0], x7_0[0]])
    ub = np.array([x_lead_0[1], v_lead_0_i[1], acc_lead_0[1], x_ego_0[1], v_ego_0[1], acc_ego_0[1], x7_0[1]])
    S = Star(lb, ub)
    mu = 0.5*(S.pred_lb + S.pred_ub)
    a = 2.5 # coefficience to adjust the distribution
    sig = (mu - S.pred_lb)/a
    Sig = np.diag(np.square(sig))
    I1 = ProbStar(S.V, S.C, S.d, mu, Sig, S.pred_lb, S.pred_ub)
    initSets.append(I1)
```

3.3 Specifying the unsafe property

After constructing an NNCS and initial set of states, users need to specify the property of the system 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 $S \triangleq Gy \leq g$, where y is the output vector of the system. Let S be an unsafe region, if the reachable sets of the system reach the unsafe region, the system is unsafe. Otherwise, it is safe. In this example, we can derive the unsafe property of the system from Eq. 6.1.

```
# unsafe constraints
# safety property: actual distance > alpha * safe distance
# <=> d = (x1 - x4) > alpha * d_safe = alpha * (1.4 * v_ego + 10)
# unsafe region: x1 - x4 <= alpha * (1.4 * v_ego + 10)

alpha = 1.0
unsafe_mat = np.array([[1.0, 0., 0., -1., -alpha*1.4, 0., 0.]])
unsafe_vec = np.array([alpha*10.0])
```

```
Unsafe matrix:
[[ 1.   0.   0.  -1.  -1.4  0.   0. ]]
Unsafe vector:  [10.]
```

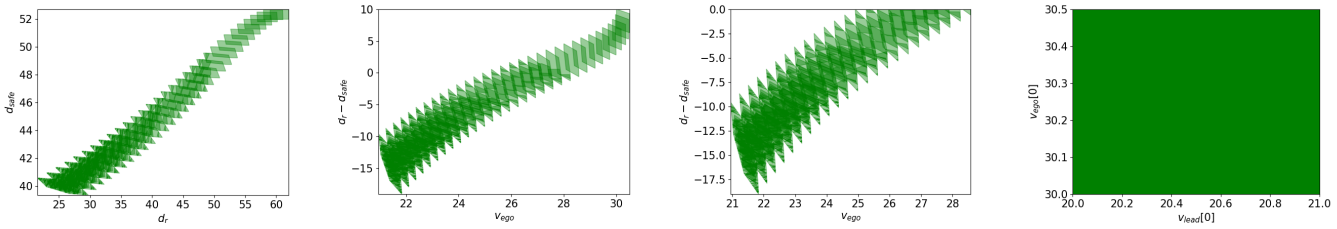
3.4 Verifying the ACC system

StarV supports automatic qualitative and quantitative verification of continuous/discrete linear NNCS. By changing pf, users can choose to conduct exact or over-approximate quantitative verification. For an NNCS, StarV computes the controller's reachable sets and then feeds them to the plant as input sets. The plant's reachable sets are then computed and fed back to the controller. To reduce conservativeness in the reachable set computation, StarV always computes the exact reachable sets for the controller (assumes it has piecewise linear activation functions). Therefore, users can choose to use parallel computing to speed up the computation.

To visualize the reachable sets of a linear NNCS, users can use the `plot_probstar_reachset` method, which plots the reachable set of the system in a specific direction defined by the *mapping matrix* and *offset vector*.

```
# ... Constructing the ACC system ...

# ... Constructing the Initial Set of States ...

# ... Specifying the unsafe property ...

# verify the system

# reference inputs
refInputs = np.array([30., 1.4])

# --- For exact qualitative and quantitative verification ---
pf = 0.0
# --- For over-approximate qualitative and quantitative verification ---
# pf = 0.001

initSet_id=5
numSteps=30
numCores=4

# verify parameters
verifyPRM = VerifyPRM_NNCS()
verifyPRM.initSet = copy.deepcopy(initSets[initSet_id])
verifyPRM.refInputs = copy.deepcopy(refInputs)
verifyPRM.numSteps = numSteps
verifyPRM.pf = pf
verifyPRM.numCores = numCores
verifyPRM.unsafeSpec = [unsafe_mat, unsafe_vec]

res = verifyBFS_DLNNCS(sys, verifyPRM)
RX = res.RX
Ce = res.CeIn
Co = res.CeOut

n = len(RX)
CE = []
CO = []
for i in range(0,n):
```

```
    if len(Ce[i]) > 0:
        CE.append(Ce[i])  # to plot counterexample set
        CO.append(Co[i])

# plot reachable set  (d_actual - d_safe) vs. (v_ego)
dir_mat1 = np.array([[0., 0., 0., 0., 1., 0., 0.],
                     [1., 0., 0., -1., -1.4, 0., 0.]])
dir_vec1 = np.array([0., -10.])

# plot reachable set  d_rel vs. d_safe
dir_mat2 = np.array([[1., 0., 0., -1., 0., 0., 0.],
                     [0., 0., 0., 0., 1.4, 0., 0.]])
dir_vec2 = np.array([0., 10.])

# plot counter input set
dir_mat3 = np.array([[0., 1., 0., 0., 0., 0., 0.],
                     [0., 0., 0., 0., 1., 0., 0.]])
dir_vec3 = np.array([0., 0.])

print('Plot reachable set...')
fig1 = plt.figure()
plot_probstar_reachset(RX, dir_mat=dir_mat2, dir_vec=dir_vec2, show_prob=False, label=('$d_{r}$','$d_{safe}$'), show=True)

fig2 = plt.figure()
plot_probstar_reachset(RX, dir_mat=dir_mat1, dir_vec=dir_vec1, show_prob=False, label=('$v_{ego}$','$d_r - d_{safe}$'), show=True)

print('Plot counter output set...')
fig3 = plt.figure()
plot_probstar_reachset(CO, dir_mat=dir_mat1, dir_vec=dir_vec1, show_prob=False, label=('$v_{ego}$','$d_r - d_{safe}$'), show=True)

print('Plot counter init set ...')
fig4 = plt.figure()
plot_probstar_reachset(CE, dir_mat=dir_mat3, dir_vec=dir_vec3, show_prob=False, label=('$v_{lead}[0]$','$v_{ego}[0]$'), show=True)
```



(a) D_r vs. D_{safe}. (b) $D_r - D_{safe}$ vs. v_{ego}. (c) Counter state (output) sets. (d) Counter initial sets.

Figure 6.2: 30-step reachable sets of ACC system controlled by $N_{5\times20}$ network. The ACC system is unsafe as $D_r - D_{safe} < 0$ from step 9. The controller does not aggressively reduce its speed to maintain safety, and the entire input set is unsafe.

Chapter 7

ProbStar Temporal Logic (ProbStarTL)

Probstar Temporal Logic (ProbStarTL), a formalism enabling quantitative verification of temporal properties of systems, which is defined on a (bounded-time) ProbStar signal (or ProbStar trace), a sequence of discrete, timed probabilistic star reachable sets. The interpretation of ProbStarTL captures a symbolic representation of the set of system traces that satisfy the specification. We currently support two basic temporal operators: always ($\Box$) and eventually ($\Diamond$). The until (U) operator is evaluated using the equivalent formula composed of the always ($\Box$) and eventually ($\Diamond$) operators.

1 Quantitative verification of Learning Enabled System (LES)

In this section, we will continue to use the learning-based Adaptive Cruise Control (ACC) system (Example 3) to illustrate how to conduct quantitative verification of the LES system using ProbStarTL in StarV [8]. Specifically, to verify the temporal behavior of the ACC system, we are interested in the discrete-time linear plant model with time bound in the system.

1.1 Example: Verification of ACC System

As shown in example 3, we first construct the ACC system and initial set of states in Example 3.1 and 3.2. Instead of specifying unsafe property as shown in Example 3.3, we create some temporal specifications for verifying the temporal behavior of the system up to T discrete-time steps. Finally, we perform quantitative verification using the user-define temporal specifications in form of ProbStarTL.

Creating Temporal Specifications

```
# load temporal and logic operator
T = 10
EVOT = _EVENTUALLY_(0,T)
lb = _LeftBracket_()
rb = _RightBracket_()
AWOT = _ALWAYS_(0,T)

# create atomic predicates
A1 = np.array([1., 0., 0., -1., -1.4, 0., 0.])
b1 = np.array([10.])
P1 = AtomicPredicate(A1,b1)

A2 = np.array([-1., 0., 0., 1., 1.4, 0., 0.])
b2 = np.array([-10.])
P1c = AtomicPredicate(A2,b2)

# convert atomic predicate to temporal specification
# phi1 : eventually_[0, T](x_lead - x_ego <= D_safe = 10 + 1.4
↪  v_ego) : A1x <= b1
phi1 = Formula([EVOT, lb, P1, rb])
print(phi1)
# phi1c : always_[0, T] (x_lead - x_ego >= D_safe = 10 + 1.4
↪  v_ego) : A2x <= b2
phi1c = Formula([AWOT, lb, P1c, rb])
print(phi1c)
```

```
# print(phi1)
Formula:
Formula type: ConjunctiveEventually
Formula length: 4
EVENTUALLY_[0,10]
(
1.0*x[0]+0.0*x[1]+0.0*x[2]-1.0*x[3]-1.4*x[4]+0.0*x[5]+0.0*x[6] <=
↪  10.0
)

# print(phi1c)
Formula:
Formula type: ConjunctiveAlways
Formula length: 4
ALWAYS_[0,10]
(
-1.0*x[0]+-0.0*x[1]+-0.0*x[2]+1.0*x[3]+1.4*x[4]+-0.0*x[5]+-0.0*x[6]
↪  <= -10.0
)
```

```

specs = [phi1, phi1c]
```

Verifying the System

```
# ... Constructing the ACC System ...

# ... Constructing the Initial Set of States ...

# ... Creating Temporal Specifications ...

# verify the system

# reference inputs
refInputs = np.array([30., 1.4])

# --- For exact qualitative and quantitative verification ---
pf = 0.0
# --- For over-approximate qualitative and quantitative verification ---
# pf = 0.001

initSet_id = 5
numSteps = T
numCores = 4

# verify parameters
verifyPRM = VerifyPRM_NNCS()
verifyPRM.initSet = copy.deepcopy(initSets[initSet_id])
verifyPRM.refInputs = copy.deepcopy(refInputs)
verifyPRM.numSteps = numSteps
verifyPRM.pf = pf
verifyPRM.numCores = numCores
verifyPRM.temporalSpecs = copy.deepcopy(specs)

traces, p_SAT_MAX, p_SAT_MIN, reachTime, checkingTime, verifyTime = verify_temporal_specs_DLNNCS(sys, verifyPRM)

print('Number of traces = {}'.format(len(traces)))
print('p_SAT_MAX = {}'.format(p_SAT_MAX))
print('p_SAT_MIN = {}'.format(p_SAT_MIN))
print('reachTime = {}'.format(reachTime))
print('checkingTime = {}'.format(checkingTime))
print('verifyTime = {}'.format(verifyTime))
```

```
Number of traces = 2
p_SAT_MAX = [0.0031680054290504113, 0.9483986782432469]
p_SAT_MIN = [0.0031680054290504113, 0.9483986782432469]
reachTime = 0.45766687393188477
checkingTime = [0.1002969741821289, 0.07552409172058105]
verifyTime = [0.5579638481140137, 0.5331909656524658]
```

2 Quantitative verification of Linear system

In this section, we are interested in discrete-time, linear time-invariant (LTI) systems with bounded time, and we will use a Timed Harmonic Oscillator example to show how we quantitatively verify the temporal properties.

2.1 Example: Verification of Timed Harmonic Oscillator System

Example 6. *The timed harmonic oscillator is defined by the system dynamics* $\dot{x} = y$, $\dot{y} = -x$, *which is a two-dimensional system. The initial state are* $x_0 \in [-6, -5]$, *and* $y_0 \in [0, 1]$. *We transform the two state variables system* $x = (x, y)^T$ *into a four-variable system* $x = (x, y, u_1, u_2)^T$ *where* u_1, u_2 *stand for system input and keep constant at each time step, and* $u_1 = u_2 = 0.5$ *for the initial state. Thus, the system dynamics are now captured by the transformed linear time-invariant system* $\dot{x} = Ax$, *where* A *is a 4-by-4 matrix:*

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Constructing the Initial Set of States

```
# system dynamics
A = np.array([[0,1,1,0],[-1,0,1,0],[0,0,0,0],[0,0,0,0]])

# input sets
init_state_bounds_list = []
dims = A.shape[0]
for dim in range(dims):
    if dim == 0:
        lb = -6
        ub = -5
    elif dim == 1:
        lb = 0
        ub = 1
    elif dim == 2:
        lb = 0.5
        ub = 0.5
    elif dim == 3:
        lb = 0.5
        ub = 0.5
    init_state_bounds_list.append((lb, ub))
init_state_bounds_array = np.array(init_state_bounds_list)
init_state_lb = init_state_bounds_array[:, 0]
init_state_ub = init_state_bounds_array[:, 1]
print(init_state_lb)
print(init_state_ub)

# construct satr set using state lower and upper bound
X0 = Star(init_state_lb,init_state_ub)

# construct probsatr set
mu_U = 0.5*(X0.pred_lb + X0.pred_ub)
a  = 3
sig_U = (X0.pred_ub-mu_U )/a
epsilon = 1e-10
sig_U = np.maximum(sig_U, epsilon)
Sig_U = np.diag(np.square(sig_U))
X0_probstar = ProbStar(X0.V, X0.C, X0.d,mu_U,
↪  Sig_U,X0.pred_lb,X0.pred_ub)
print(X0_probstar)
```

```
# print(init_state_lb)
[-6.   0.   0.5  0.5]

# print(init_state_ub)
[-5.   1.   0.5  0.5]

# print(X0_probstar)
ProbStar Set:
V: [[-5.5  0.5  0. ]
 [ 0.5  0.   0.5]
 [ 0.5  0.   0. ]
 [ 0.5  0.   0. ]]
Predicate Constraints:
C: []
d: []
dim: 4
nVars: 2
pred_lb: [-1. -1.]
pred_ub: [1. 1.]
mu: [0. 0.]
Sig: [[0.11111 0.     ]
 [0.      0.11111]]
```

Creating Temporal Specifications

```
# load temporal and logic operator
h = math.pi/4
time_bound = math.pi
T = int (time_bound/ h)
AND = _AND_()
OR = _OR_()
lb = _LeftBracket_()
rb = _RightBracket_()
EVOT =_EVENTUALLY_(0,T)
EVOT1 =_EVENTUALLY_(2,3)
AWOT = _ALWAYS_(0,T)
AWOT1 = _ALWAYS_(1,2)

# create atomic predicates
A1 = np.array([-1., 0.])
b1 = np.array([-4])
P1 = AtomicPredicate(A1,b1)
A2 = np.array([1,0])
b2 = np.array([4])
P2 = AtomicPredicate(A2,b2)

# convert atomic predicate to temporal specification
# phi1 : eventually_[0, T](x >= 4) : A1x <= b1
phi1 = Formula([EVOT,P1])
```

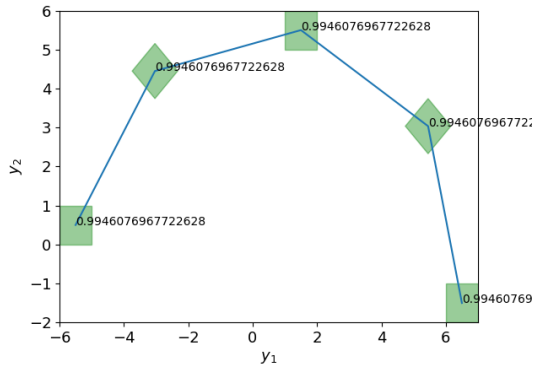
```
# print(phi1)
Formula:
Formula type: ConjunctiveEventually
Formula length: 2
EVENTUALLY_[0,4]
-1.0*x[0]+0.0*x[1] <= -4

# print(phi2)
Formula:
Formula type: DisjunctiveAlways
Formula length: 9
ALWAYS_[1,2]
(
1*x[0]+0*x[1] <= 4
OR
(
EVENTUALLY_[2,3]
-1.0*x[0]+0.0*x[1] <= -4
)
)
```

```
print(phi1)
# phi2 : always_[1,2](x <= 4 OR eventually_[2, 3](x >=4))
phi2 = Formula([AWOT1,lb,P2,OR,lb,EVOT1,P1,rb,rb])
print(phi2)

specs = [phi1,phi2]
```

Verifying the System

```
# ... Constructing the Initial Set of States ...

# ... Conducting Reachability Analysis...

# ... Creating Temporal Specifications ...

# verify the system

# set paprameter for reachability
m = 2
target_error = 1e-6
samples = 51
tolerance = 1e-9
use_arnoldi = True
use_init_space = False
output_space = np.array([[1,0,0,0],[0,1,0,0]])
initial_space = X0_probstar.V

reach_start_time = time.time()
R,_= simReachKrylov(A,X0_probstar,h, T,
↪  m,samples,tolerance,target_error = target_error,initial_space
↪  = initial_space,output_space = output_space,use_arnoldi =
↪  use_arnoldi,use_init_space = use_init_space)
reach_time = time.time() - reach_start_time

reachTime = []
checkingTime = []
verifyTime = []
p_SAT_MAX = []
p_SAT_MIN= []

for i in range(0,len(specs)):
    check_start = time.time()
    spec = specs[i]
    DNF_spec = spec.getDynamicFormula()
    _,p_max, p_min,_= DNF_spec.evaluate(R)
    end = time.time()
    reachTime.append(reach_time)
    check_time = end -check_start
    checkingTime.append(check_time)
    verifyTime.append(check_time + reach_time)
    p_SAT_MAX.append(p_max)
    p_SAT_MIN.append(p_min)


print('p_SAT_MAX = {}'.format(p_SAT_MAX))
print('p_SAT_MIN = {}'.format(p_SAT_MIN))
print('reachTime = {}'.format(reachTime))
print('checkingTime = {}'.format(checkingTime))
print('verifyTime = {}'.format(verifyTime))

plot_probstar_signal(R)
```

```
p_SAT_MAX = [0.9931807445107577, 0.9969978743987328]
p_SAT_MIN = [0.9931807445107577, 0.9969978743987328]
reachTime = [0.005526304244995117, 0.005526304244995117]
checkingTime = [0.015836000442504883, 4.166131973266602]
verifyTime = [0.0213623046875, 4.171658277511597]
```



plot_probstar_signal(R)

Bibliography

[1] Stanley Bak and Parasara Sridhar Duggirala. Simulation-equivalent reachability of large linear systems with inputs. In *International Conference on Computer Aided Verification*, pages 401–420. Springer, 2017.

[2] Ryan Brown, Luan Viet Nguyen, Weiming Xiang, Marilyn Wolf, and Hoang-Dung Tran. Perception-based quantitative runtime verification for learning-enabled cyber-physical systems. In *2025 the 16th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages –. ACM/IEEE, 2025.

[3] Parasara Sridhar Duggirala and Mahesh Viswanathan. Parsimonious, simulation based verification of linear systems. In *International Conference on Computer Aided Verification*, pages 477–494. Springer, 2016.

[4] Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks. *arXiv preprint arXiv:1706.07351*, 2017.

[5] Open Neural Network Exchange (ONNX).

[6] Apala Pramanik, Sung Woo Choi, Yuntao Li, Luan Viet Nguyen, Kyungki Kim, and Hoang-Dung Tran. Perception-based runtime monitoring and verification for human-robot construction systems. In *2024 22nd ACM-IEEE International Symposium on Formal Methods and Models for System Design (MEMOCODE)*, pages 124–134. IEEE, 2024.

[7] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):41, 2019.

[8] Hoang-Dung Tran, Sungwoo Choi, Yuntao Li, Hideki Okamoto, Bardh Hoxha, and Georgios Fainekos. Probstar temporal logic for verifying complex behaviors of learning-enabled systems. In *Proceedings of the 28th ACM International Conference on Hybrid Systems: Computation and Control*, 2025.

[9] Hoang-Dung Tran, Sungwoo Choi, Hideki Okamoto, Bardh Hoxha, Georgios Fainekos, and Danil Prokhorov. Quantitative verification for neural networks using probstars. In *Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control*, pages 1–12, 2023.

[10] Hoang-Dung Tran, Patrick Musau, Diego Manzanas Lopez, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. Star-based reachability analsysis for deep neural networks. In *23rd International Symposisum on Formal Methods (FM'19)*. Springer International Publishing, October 2019.

[11] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T Johnson. Nnv: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In *International Conference on Computer Aided Verification*, pages 3–17. Springer, 2020.

[12] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. In *Advances in Neural Information Processing Systems*, pages 4944–4953, 2018.