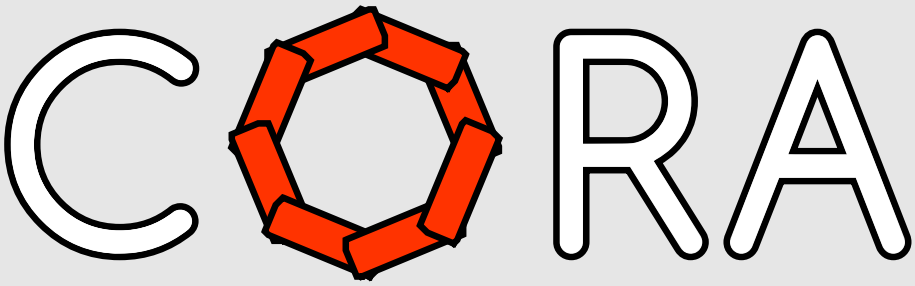
# Manual

v2025.1.0




**Matthias Althoff, Niklas Kochdumper,
Tobias Ladner, and Mark Wetzlinger**

Technische Universität München
85748 Garching, Germany

**Abstract**

The Continuous Reachability Analyzer (CORA) is a MATLAB-based toolbox designed for the formal verification of cyber-physical systems through reachability analysis. It offers a comprehensive suite of tools for modeling and analyzing various system dynamics, including linear, nonlinear, and hybrid systems. CORA supports both continuous and discrete-time systems, accommodating uncertainties in system inputs and parameters. These uncertainties are captured by a diverse range of set representations such as intervals, zonotopes, Taylor models, and polytopes. Additionally, CORA provides functionalities for the formal verification of neural networks as well as data-driven system identification with reachset conformance. Various converters are implemented to easily model a system in CORA such as the well-established SpaceEx format for dynamic systems and ONNX format for neural networks. CORA ensures the seamless integration of different reachability algorithms without code modifications and aims for a user-friendly experience through automatic parameter tuning, making it a versatile tool for researchers and engineers in the field of cyber-physical systems.

# Contents

# 1 Introduction

This section shortly introduces the main concepts of the CORA toolbox, provides detailed instructions for the installation, and summarizes the connections of CORA to other tools.

## 1.1 What's new compared to CORA v2024?

It is our pleasure to present many new features for CORA v2025.1.0. The subsequent list is non-exhaustive and unsorted:

- **New reachability algorithms:** We have added a novel set-based reachability algorithm for computing inner approximations of nonlinear systems [1]. Furthermore, we have integrated recently proposed backward reachability algorithms for linear continuous-time systems [2].
- **New spectrahedral shadow class:** We added a new set representation `spectraShadow` to CORA (Sec. 2.2.1.10): Spectrahedral shadows can be seen as the semidefinite generalization of polytopes, and can represent a large variety of convex sets. In particular, every convex set representation implemented in CORA can be represented as a spectrahedral shadow.
- **Improved neural network verification and new set-based training:** We improved the code to verify neural networks and harmonized it with all other CORA modules. Additionally, we added a novel approach to train verifiably robust networks both in the supervised [3] and in the reinforcement learning [4] setting. Please visit Sec. 6 for more information.
- **Reachset-conformance identification:** We added several new functionalities to the reachset-conformance identification in CORA. Please visit Sec. 4.1.10 and [5,6] for details.
- **Verification of dynamic systems using signal temporal logic:** Improvements have been made how signal temporal logic can be used to verify dynamic systems (Tab. 29), in particular using a new incremental algorithm to verify hybrid systems [7].
- **Overhaul of dynamic systems:** We overhauled all dynamics system classes to ensure the correctness of our implementation and improve maintainability. This also comes with an updated syntax for hybrid systems as described in Sec. 4.3, in particular how reset functions are constructed.
- **Polytopes:** Our `polytope` class got a massive overhaul with several new functionalities [8]. Vertex representation and halfspace representation can now also be instantiated independently from another, making the usage more flexible. Visit Sec. 2.2.1.4 for details.
- **Matrix sets:** We improved the performance of matrix sets by restructuring their properties as multi-dimensional matrices instead of cell arrays. Please visit Sec. 3 for the new syntax.
- **Improved `contains` method:** The method `contains` now also provides a certificate whether the result could be certified. While this was always the case when the method returned `true`, the inverse cannot always be certified due to outer approximations. See Sec. 2.1.2.1 for more information.
- **Miscellaneous:** Minor improvements have been made in various parts of the code: Some basic functionality runs more efficiently, standardized input argument validation and more accurate error messages enhance the responsiveness, and more unit tests ensure greater reliability. Please also have a look at Appendix A.12 for deprecated functionality and their replacements.

## 1.2 Philosophy

The **CO**ntinuous **R**eachability **A**nalyzer (CORA)[1] is a MATLAB toolbox for prototypical design of algorithms for reachability analysis. The toolbox is designed for various kinds of systems with purely continuous dynamics (linear systems, nonlinear systems, differential-algebraic systems, parameter-varying systems, etc.) and hybrid dynamics combining the aforementioned continuous dynamics with discrete transitions. Let us denote the continuous part of the solution of a hybrid system for a given initial discrete state by $\chi(t; x_0, u(\cdot), p)$, where $t \in \mathbb{R}$ is the time, $x_0 \in \mathbb{R}^n$ is the continuous initial state, $u(t) \in \mathbb{R}^m$ is the system input at $t$, $u(\cdot)$ is the input trajectory, and $p \in \mathbb{R}^p$ is a parameter vector. The continuous reachable set at time $t = t_f$ can be defined for a set of initial states $\mathcal{X}_0$, a set of input values $\mathcal{U}(t)$, and a set of parameter values $\mathcal{P}$, as

$$\mathcal{R}^e(t_f) = \Big\{ \chi(t_f; x_0, u(\cdot), p) \in \mathbb{R}^n \big| x_0 \in \mathcal{X}_0, \forall t : u(t) \in \mathcal{U}(t), p \in \mathcal{P} \Big\}.$$

CORA mainly supports over-approximative computation of reachable sets since (a) exact reachable sets cannot be computed for most system classes [9] and (b) over-approximative computations allow for safety verification. Thus, CORA computes over-approximations for particular points in time $\mathcal{R}(t) \supseteq \mathcal{R}^e(t)$ and for time intervals: $\mathcal{R}([t_0, t_f]) = \bigcup_{t \in [t_0, t_f]} \mathcal{R}(t)$.

CORA also enables the construction of an individual reachable set computation in a relatively short amount of time. This is achieved by the following design choices:

- CORA is programmed in MATLAB, which is a script-based programming environment. Since the code does not have to be compiled, one can stop the program at any time and directly see the current values of variables. This makes it especially easy to understand the workings of the code and to debug new code.
- CORA is an object-oriented toolbox that uses modularity, operator overloading, inheritance, and information hiding. One can safely use existing classes and just adapt classes of interest without redesigning the whole code. Operator overloading facilitates writing formulas that look almost identical to the ones derived in scientific papers and thus reduces programming errors. Most of the information of each class is hidden and not relevant to users of the toolbox. We try to haromize the syntrax of methods so that set representations and dynamic systems can be effortlessly replaced.

Of course, it is also possible to use CORA as it is, to perform reachability analysis.

> Please be aware of the fact that outcomes of reachability analysis heavily depend on the chosen parameters for the analysis (those parameters are listed in Sec. 4.1.1). Improper choice of parameters can result in an unacceptable over-approximation although reasonable results could be achieved by using appropriate parameters. Thus, self-tuning of the parameters for reachability analysis, as it is already done by the adaptive algorithm for linear and nonlinear systems, is investigated as part of ongoing and future work.

Since this manual focuses on the presentation of the capabilities of CORA, no other tools for reachability analysis of continuous and hybrid systems are reviewed. A list of related tools is presented in [10–12].

[1] Website: `https://cora.in.tum.de/`

## 1.3 Installation

We provide a script to guide you through the installation process. To run it, download the CORA repository[2], add CORA to the MATLAB path, and type `installCORA` in the command window. Some toolbox installations require a restart of MATLAB, and you have to manually rerun the installation script afterward.

CORA requires the following MATLAB toolboxes:

- **Symbolic Math Toolbox**
- **Optimization Toolbox**
- **Statistics and Machine Learning Toolbox**
- (Optional) **Multiple precision toolbox** from the Mathworks File Exchange (only required for Krylov sub-space methods)

In addition, CORA uses the following third-party toolboxes that have to be installed:

- **YALMIP**: The YALMIP toolbox [13] is designed for solving optimization problems of various types. CORA requires the YALMIP toolbox along with at least one supported Semi-Definite Program (SDP) solver.
- (Optional) **MOSEK**[3], **SDPT3**[4]: To use the ellipsoid set representation more efficiently, consider installing either the MOSEK or SDPT3 solver. Note: If you want to use MOSEK, please make sure that you do not override the built-in MATLAB optimization routines (`linprog`, `quadprog`, etc.), as this will break certain parts of CORA.

The installation of all required toolboxes can be checked individually by running `test_requiredToolboxes`. To check whether the core functionality of CORA has been set up correctly, run the standard test suite `runTestSuite` which should take about 5 minutes.

For the verification of neural networks (Sec. 4.2.11 and Sec. 6), the following toolboxes are required:

- **Deep Learning Toolbox**
- **Deep Learning Toolbox Converter for ONNX Model Format**
- **Parallel Computing Toolbox** (enables GPU usage)

These toolboxes can also be installed via the installation script. To check whether the neural network functionality has been set up correctly, run `testnn_requiredToolboxes` or the neural network test suite `runTestSuite('nn')`.

Please visit our website to get started: `https://cora.in.tum.de`

## 1.4 Connections to and from SpaceEx

As part of the EU project Unifying Control and Verification of Cyber-Physical Systems (UnCoVerCPS) the tools CORA and SpaceEx [14] have been integrated to a certain extent.

[2]Website: `https://cora.in.tum.de`: If you use git to clone the CORA repository, please also download larger data files using git lfs (large file storage): i) Install git lfs (`https://git-lfs.com/`) and ii) run `git lfs pull` to ensure all data files are downloaded correctly.

[3]`https://www.mosek.com/`

[4]`https://blog.nus.edu.sg/mattohkc/softwares/sdpt3/`

#### Importing and Exporting SpaceEx Models

CORA can read SpaceEx models as described in Sec. 8 and CORA models can be exported as SpaceEx models as detailed in Sec. 4.1.11. This has two major benefits: First, SpaceEx has become the quasi-standard for model exchange between tools for formal verification of hybrid systems (see ARCH friendly competition in Sec. 1.5) so that many model files in this format are available. Second, SpaceEx offers a graphical model editor which is briefly presented in Sec. 8.1, helping non-experts to easily model hybrid systems.

#### CORA/SX

CORA code for computing reachable sets of nonlinear systems is available in the SpaceEx extension CORA/SX as C++ code. CORA has several implementations to compute reachable sets of nonlinear systems—in the first CORA/SX version, the most basic, but very efficient algorithm from [15] has been implemented. Also, the zonotope class from CORA is available in CORA/SX, making efficient computations for switched linear systems possible as described in [16].

## 1.5 CORA@ARCH

The ARCH[5] friendly competition is the main platform for comparing the results of different reachability tools on multiple challenging benchmark problems. CORA has participated in the ARCH friendly competitions since the first competition in 2017. Results of the competition can be found in the yearly ARCH proceedings [17–19]. In particular, CORA has participated in the linear systems category [20–23] and the nonlinear systems category [24–27]; CORA/SX has participated in the same categories in 2018 [21, 25] and in the linear systems category in 2019 [22].

All results from all tools participating in the friendly competitions can be re-computed using the ARCH repeatability packages, which are publicly available: gitlab.com/goranf/ARCH-COMP/.

The results from the last ARCH competition can be found in the CORA toolbox at *examples/ARCHcompetition/*. We also published the results as Code Ocean capsules[6], which allows everyone to conveniently reproduce the results online without the need to install anything.

More information on the ARCH workshops can be found here: cps-vo.org/group/ARCH.

## 1.6 Architecture

The architecture of CORA can essentially be grouped into the parts presented in Fig. 1 using a UML[7] class diagram: Classes for set representations (Sec. 2), classes for matrix set representations (Sec. 3), classes for the analysis of continuous dynamics (Sec. 4.2), classes for the analysis of hybrid dynamics (Sec. 4.3), and classes for the abstraction to discrete systems (Sec. 5).

All classes for set representations inherit some common properties and functionality from the parent class `contSet` (see Fig. 1). Similary, all classes for continuous dynamics inherit from the parent class `contDynamics` (see Fig. 1).

For hybrid systems, the class diagram in Fig. 1 shows that parallel hybrid automata (class `parallelHybridAutomaton`) consist of several instances of hybrid automata (class `hybridAutomaton`), which in turn consist of several instances of the `location` class. Each

[5] Applied Verification for Continuous and Hybrid Systems
[6] see `https://codeocean.com/capsule/2113947/tree` and `https://codeocean.com/capsule/1267711/tree`
[7] `http://www.uml.org/`

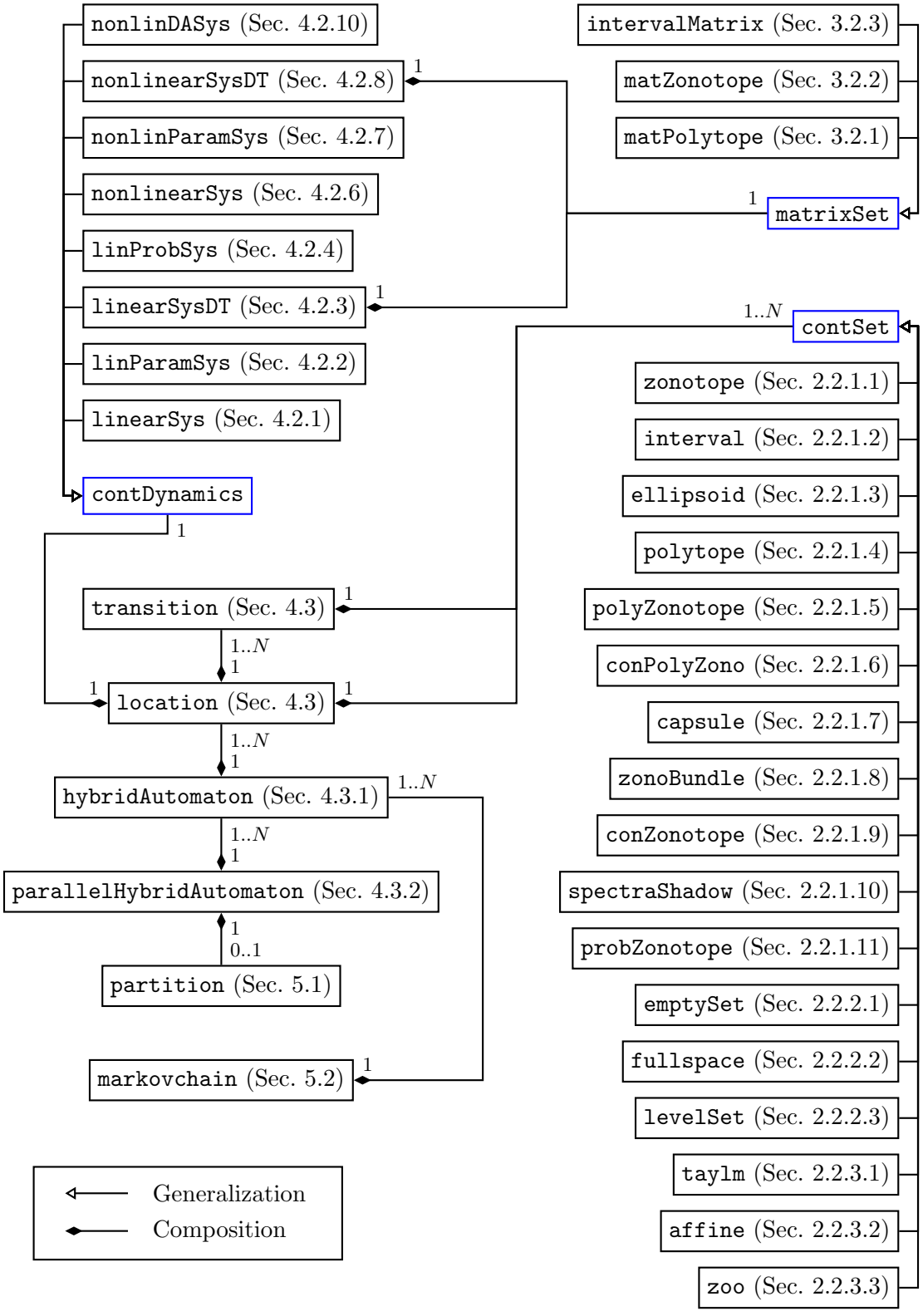



Figure 1: Unified Modeling Language (UML) class diagram of CORA.

`location` object has continuous dynamics (classes inheriting from `contDynamics`), several transitions (class `transition`), and a set representation (classes inheriting from `contSet`) to describe the invariant of the location. Each transition has a set representation to describe the guard set enabling a transition to the next discrete state. More details on the semantics of those components can be found in Sec. 4.3.

Note that some classes subsume the functionality of other classes. For instance, nonlinear differential-algebraic systems (class `nonlinDASys`) are a generalization of nonlinear systems (class `nonlinearSys`). Less general systems are not removed because very efficient algorithms exist for those systems that are not applicable to more general systems.

## 1.7 Unit Tests

To ensure that all functions in CORA work as they should, CORA contains a number of unit tests in the folder `unitTests`. Those unit tests are executed by different test suits:

- `runTestSuite`: This test suite should always be executed after installing CORA or updating MATLAB/CORA and runs all basic tests. All files whose function name starts with `test_` are executed.
- `runTestSuite` also takes a parameter `testSuite` specifying whether certain components of CORA should be tested more sophistically. Possible values are `'short'` (default), `'long'`, `'nn'`, `'examples'`, `'benchmark'`, and `'website'`. Note: Some of these test suites take a long time to finish.

Additionally, we provide some unit tests for third-party packages:

- `runTestSuite_INTLAB`: This test suite compares the interval arithmetic results with those of INTLAB[8]. To successfully execute those tests, INTLAB has to be installed. The tests are randomized and for each function, thousands of samples are generated. Simple, non-randomized tests for interval arithmetic are already included in `runTestSuite`. This test suite executes all files whose function name starts with `testINTLAB_`.
- `runTestSuite_Mosek`: This test suite runs all files requiring the Mosek solver[9], i.e., those that start with `testMosek_`.
- `runTestSuite_MP`: This test suite runs all files requiring the multiple precision toolbox[10], i.e., those that start with `testMP_`.
- `runTestSuite_SDPT3`: This test suite runs all files requiring the SDPT3 solver[11], i.e., those that start with `testSDPT3_`.

**Note:** According to our experience, results may vary numerically depending on the installed MATLAB version. CORA v2025.1.0 has been tested using MATLAB R2024b.

[8] `http://www.ti3.tu-harburg.de/intlab/`
[9] `https://www.mosek.com`
[10] `https://www.mathworks.com/matlabcentral/fileexchange/6446-multiple-precision-toolbox-for-matlab`
[11] `https://blog.nus.edu.sg/mattohkc/softwares/sdpt3`

# 2 Set Representations and Operations

This section introduces the set representations and set operations that are implemented in the CORA toolbox.

## 2.1 Set Operations

The reachability algorithms implemented in CORA rely on set-based computation. One major design principle is that the same standard set operations are implemented for all set representations so that algorithms can be executed with different set representations. In this section, we introduce the most important set operations, which are demonstrated by examples involving concrete set representations. Set representations are later detailed in Sec. 2.2; however, in order to follow the subsequent examples, it suffices to consider the sets as arbitrary continuous sets.

If a set representation is not closed under an operation, an over-approximation is returned (see Tab. 1).

### 2.1.1 Basic Set Operations

We first consider basic operations on sets.

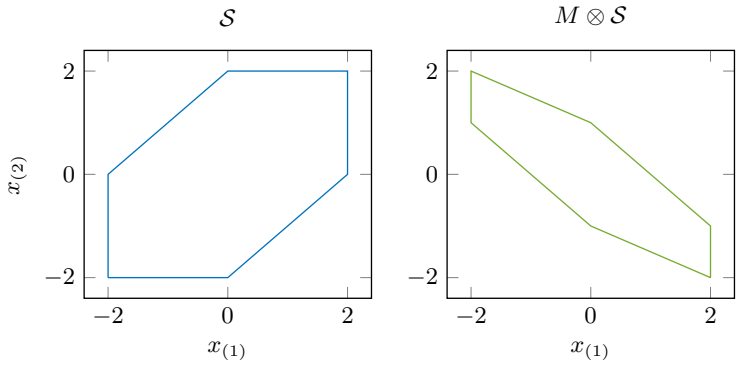
#### 2.1.1.1 mtimes

The method `mtimes`, which overloads the * operator, implements the linear map of a set. Given a set $\mathcal{S} \subset \mathbb{R}^n$, the linear map is defined as

$$\texttt{mtimes}(M, \mathcal{S}) = M \otimes \mathcal{S} = \{Ms \mid s \in \mathcal{S}\}, \; M \in \mathbb{R}^{w \times n}.$$

It is also possible to consider a set of matrices $\mathcal{M} \subset \mathbb{R}^{w \times n}$ instead of a fixed-value matrix $M \in \mathbb{R}^{w \times n}$ (see Sec. 3.1.1). Let us demonstrate the method `mtimes` by an example:

```
% set and matrix
S = zonotope([0 1 1 0; ...
              0 1 0 1]);
M = [1 0; -1 0.5];

% linear transformation
res = M * S;
```



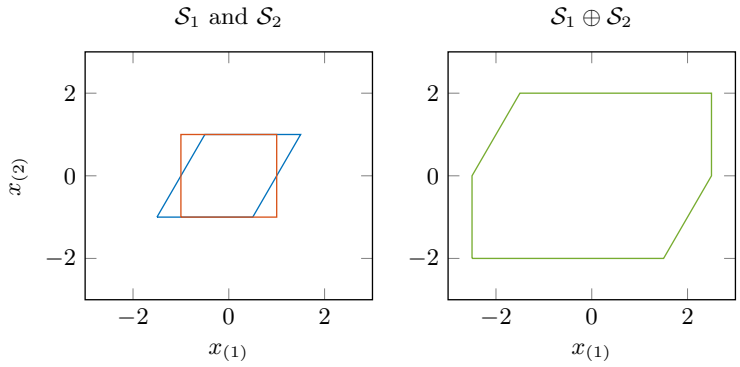

#### 2.1.1.2 plus

The method `plus`, which overloads the + operator, implements the Minkowski sum of two sets. Given two sets $\mathcal{S}_1, \mathcal{S}_2 \subset \mathbb{R}^n$, the Minkowski sum is defined as

$$\texttt{plus}(\mathcal{S}_1, \mathcal{S}_2) = \mathcal{S}_1 \oplus \mathcal{S}_2 = \{s_1 + s_2 \mid s_1 \in \mathcal{S}_1, \; s_2 \in \mathcal{S}_2\}.$$

Let us demonstrate the method `plus` by an example:

```
% set S1 and S2
S1 = zonotope([0 0.5 1; ...
               0 1   0]);
S2 = zonotope([0 1 0; ...
               0 0 1]);

% Minkowski sum
res = S1 + S2;
```




### 2.1.1.3 cartProd

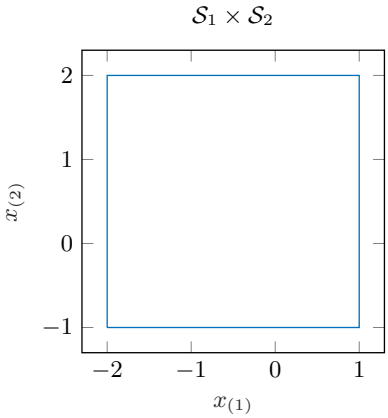
The method `cartProd` implements the Cartesian product of two sets. Given two sets $\mathcal{S}_1 \subset \mathbb{R}^n$ and $\mathcal{S}_2 \subset \mathbb{R}^w$, the Cartesian product is defined as

$$\texttt{cartProd}(\mathcal{S}_1, \mathcal{S}_2) = \mathcal{S}_1 \times \mathcal{S}_2 = \{[s_1 \; s_2]^T \mid s_1 \in \mathcal{S}_1, \; s_2 \in \mathcal{S}_2\}.$$

Let us demonstrate the method `cartProd` by an example:

```
% set S1 and S2
S1 = interval(-2,1);
S2 = interval(-1,2);

% Cartesian product
res = cartProd(S1,S2)
```

```
Command Window:

res =
[-2.00000,1.00000]
[-1.00000,2.00000]
```



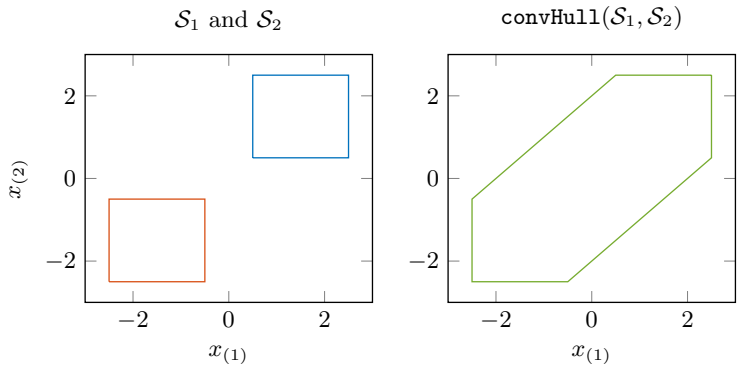

### 2.1.1.4 convHull

The method `convHull` implements the convex hull of two sets. Given two sets $\mathcal{S}_1, \mathcal{S}_2 \subset \mathbb{R}^n$, the convex hull is defined as

$$\texttt{convHull}(\mathcal{S}_1, \mathcal{S}_2) = \{\lambda s_1 + (1-\lambda) s_2 \mid s_1, s_2 \in \mathcal{S}_1 \cup \mathcal{S}_2, \; \lambda \in [0,1]\}.$$

Furthermore, given a single non-convex set $\mathcal{S} \subset \mathbb{R}^n$, $\texttt{convHull}(\mathcal{S})$ computes the convex hull of the set. Let us demonstrate the method `convHull` by an example:

```
% set S1 and S2
S1 = conZonotope([1.5 1 0; ...
                  1.5 0 1]);
S2 = conZonotope([-1.5 1 0; ..
                  -1.5 0 1]);

% convex hull
res = convHull(S1,S2);
```




#### 2.1.1.5 quadMap

The method `quadMap` implements the quadratic map of a set. Given a set $\mathcal{S} \subset \mathbb{R}^n$, the quadratic map is defined as

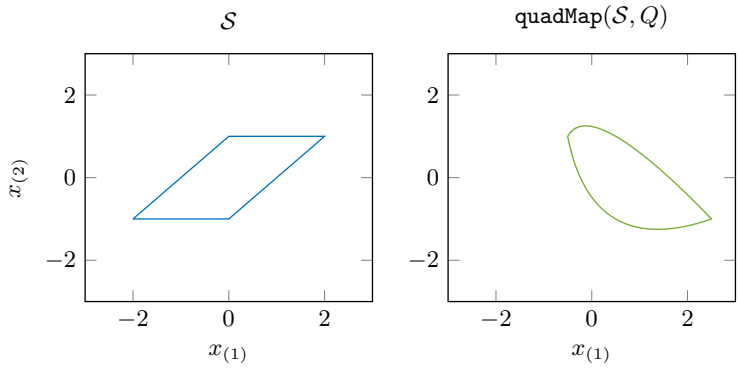
$$\texttt{quadMap}(\mathcal{S}, Q) = \{x \mid x_{(i)} = s^T Q_i s,\ s \in \mathcal{S},\ i = 1 \dots w\},\ Q_i \in \mathbb{R}^{n \times n},$$

where $x_{(i)}$ is the $i$-th value of the vector $x$. If `quadMap` is called with two different sets $\mathcal{S}_1, \mathcal{S}_2 \subset \mathbb{R}^n$ as input arguments, the method computes the mixed quadratic map:

$$\texttt{quadMap}(\mathcal{S}_1, \mathcal{S}_2, Q) = \{x \mid x_{(i)} = s_1^T Q_i s_2,\ s_1 \in \mathcal{S}_1,\ s_2 \in \mathcal{S}_2,\ i = 1 \dots w\},\ Q_i \in \mathbb{R}^{n \times n}.$$

Let us demonstrate the method `quadMap` by an example:

```
% set and matrices
S = polyZonotope([0;0], ...
                 [1 1;1 0], ..
                 [],eye(2));

Q{1} = [0.5 0.5; 0 -0.5];
Q{2} = [-1 0; 1 1];

% quadratic map
res = quadMap(S,Q);
```



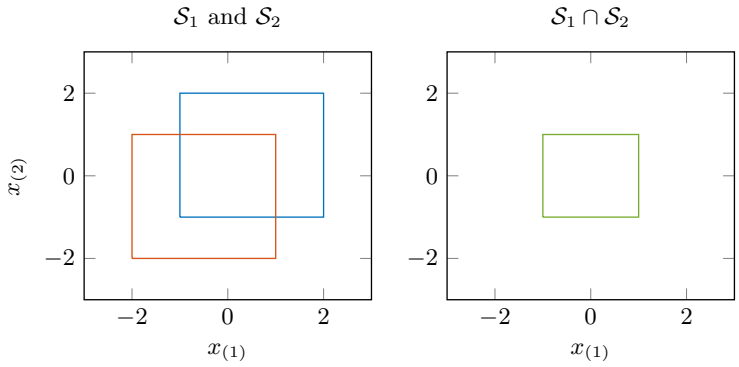

#### 2.1.1.6 and

The method `and`, which overloads the & operator, implements the intersection of two sets. Given two sets $\mathcal{S}_1, \mathcal{S}_2 \subset \mathbb{R}^n$, the intersection is defined as

$$\texttt{and}(\mathcal{S}_1, \mathcal{S}_2) = \mathcal{S}_1 \cap \mathcal{S}_2 = \{s \mid s \in \mathcal{S}_1,\ s \in \mathcal{S}_2\}.$$

Let us demonstrate the method `and` by an example:

```
% set S1 and S2
S1 = interval([-1;-1],[2;2]);
S2 = interval([-2;-2],[1;1]);

% intersection
res = S1 & S2;
```



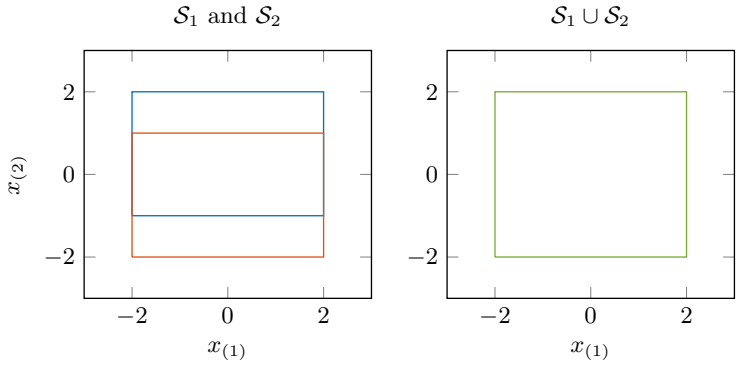

#### 2.1.1.7 or

The method `or`, which overloads the | operator, implements the union of two sets. Given two sets $\mathcal{S}_1, \mathcal{S}_2 \subset \mathbb{R}^n$, their union is defined as

$$\texttt{or}(\mathcal{S}_1, \mathcal{S}_2) = \mathcal{S}_1 \cup \mathcal{S}_2 = \{s \mid s \in \mathcal{S}_1 \vee s \in \mathcal{S}_2\}.$$

Let us demonstrate the method `or` by an example:

```
% set S1 and S2
S1 = interval([-2;-1],[2;2]);
S2 = interval([-2;-2],[2;1]);

% union
res = S1 | S2;
```




#### 2.1.1.8 minkDiff

The method `minkDiff` implements the Minkowski difference of two sets. Given two sets $\mathcal{S}_1, \mathcal{S}_2 \subset \mathbb{R}^n$, their Minkowski difference is defined as

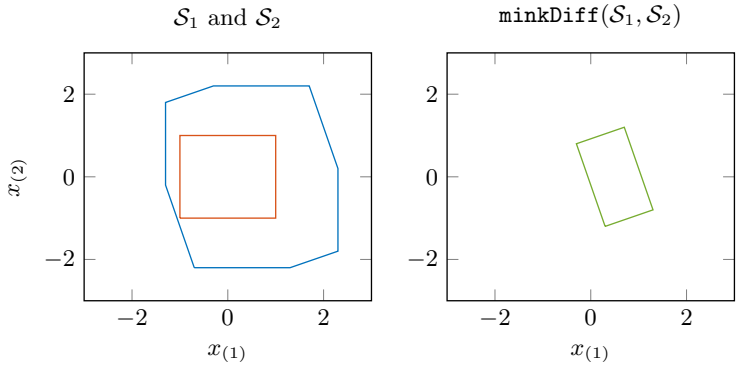
$$\texttt{minkDiff}(\mathcal{S}_1, \mathcal{S}_2) = \{s \in \mathbb{R}^n \mid s \oplus \mathcal{S}_2 \subseteq \mathcal{S}_1\}.$$

Let us demonstrate the method `minkDiff` by an example:

Table 1: Relations between set representations and set operations. The shortcuts e (exact computation) and o (over-approximation) are used. The symbol e* indicates that the operation is exact if no independent generators (see Sec. 2.2.1.5 and Sec. 2.2.1.6 for details) are used.

| Set Rep. | Lin. Map | Mink. Sum | Cart. Prod. | Conv. Hull | Quad. Map | Inter-section | Union |
|---|---|---|---|---|---|---|---|
| `interval` | o | e | e | o | | e | o |
| `zonotope` | e | e | e | o | o | o | o |
| `polytope` | e | e | e | e | | e | o |
| `conZonotope` | e | e | e | e | o | e | o |
| `zonoBundle` | e | e | e | e | o | e | o |
| `ellipsoid` | e | o | o | o | | o | |
| `capsule` | e | o | | | | | |
| `taylm` | e | e | e | | | | |
| `polyZonotope` | e | e | e | e* | e* | | |
| `conPolyZono` | e | e | e | e* | e* | e* | e* |

```
% set 1 and set 2
S1 = zonotope( ...
        [0.5 0.5 -0.3 1 0; ...
         0   0.2  1   0 1]);
S2 = interval([-1;-1],[1;1]);

% Minkowski difference
res = minkDiff(S1,S2);
```




### 2.1.2 Predicates

Predicates check if sets fulfill certain properties and return either `true` or `false`.

#### 2.1.2.1 contains

The method `contains` checks if a set is contains another set. Given two sets $\mathcal{S}_1, \mathcal{S}_2 \subset \mathbb{R}^n$, the method `contains` is defined as

$$\texttt{contains}(\mathcal{S}_1, \mathcal{S}_2) = \begin{cases} \texttt{true}, & \mathcal{S}_2 \subseteq \mathcal{S}_1, \\ \texttt{false} & \text{otherwise.} \end{cases}$$

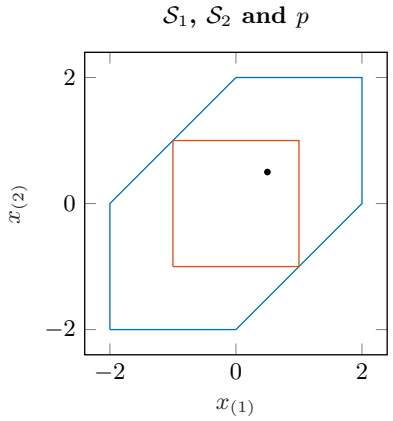
In addition, the method `contains` can be applied to check if a point or a point cloud (represented as a matrix whose columns are individual points) is located inside a set. For point clouds, we return the result of the containment check for each individual point in a matrix. Since containment checks can be computationally expensive, we implemented over-approximative algorithms for some set representations (see Tab. 2). If the over-approximative algorithm returns `true`, it is guaranteed that $\mathcal{S}_2$ is contained in $\mathcal{S}_1$. However, if the over-approximative algorithm returns `false`, the set $\mathcal{S}_2$ could still be contained in $\mathcal{S}_1$. To execute the over-approximative instead of the exact algorithm, one has to add the flag 'approx':

```
res = contains(S1,S2,'approx');
```

Let us demonstrate the method `contains` by an example:

```
% sets S1,S2, and point p
S1 = zonotope([0 1 1 0; ...
               0 1 0 1]);
S2 = interval([-1;-1],[1;1]);
p = [0.5;0.5];

% containment check
res1 = contains(S1,S2)
res2 = contains(S1,p)
```

```
Command Window:

res1 = true

res2 = true
```




For certain set representations, the method `contains` also admits additional input and output options:

```
[res, cert, scaling] = contains(S1, S2, method, tol, maxEval);
```

The optional argument `method` specifies which algorithm should be used to solve the containment problem. Apart from `'exact'` and `'approx'`, some set representations have a sampling-based algorithm (see e.g., [28]) that can be called using the option `method = 'sampling'`, or an optimization-based algorithm (see [29]) that can be called using the option `method = 'opt'`. Some set representations also have special sub-algorithms. For instance, the zonotope-in-zonotope containment problem admits two approximative algorithms, `approx:st` and `approx:stDual`. We refer to the corresponding function file for more details on available algorithms.

The optional argument `tol` indicates the tolerance for the containment check: the higher the tolerance, the more likely it is that points near the boundary of `S1` will be detected as lying in `S1`, which can be useful to counteract errors originating from floating point errors.

The optional argument `maxEval` is used for sampling- or optimization-based algorithms, and indicates the maximum number of steps that the algorithm should perform.

The output `cert` is set to `true` if and only if the result of `res` could be verified. This is particularly useful for approximative algorithms, as it allows to distinguish between cases where the algorithm either confirms containment (in which case `res = true` and `cert = true`), disproves containment (in which case `res = false` and `cert = true`), or fails to confirm or disprove containment (in which case `res` can be arbitrary, but `cert = false`). Specifically, if `cert = false`, the set `S_2` might still be contained in `S_1` even if `res = false`.

Finally, the output `scaling` is the minimal scalar such that

$$\texttt{S\_2} \subseteq \texttt{center(S\_1) + scaling * (S\_1 - center(S\_1))}.$$

The method `enlarge` (Sec. 2.1.4.4) computes this scaling in CORA. Please note that `scaling` is infinite if the subset relation above cannot be established.

Table 2: Containment checks $\mathcal{S}_2 \subseteq \mathcal{S}_1$ implemented by the method `contains`$(\mathcal{S}_1, \mathcal{S}_2)$ in CORA. The column headers represent the circumbody $\mathcal{S}_1$ and the row headers represent the set inbody $\mathcal{S}_2$. The shortcuts e (exact check) and o (over-approximation) are used. If both, an exact and an over-approximative algorithm are implemented, we write e/o.

| | **I** | **Z** | **P** | **cZ** | **zB** | **E** | **C** | **pZ** | **cPZ** | **levelSet** | **SpS** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **interval** (I) | e | e/o | e | e/o | e/o | e | e | o | o | o | o |
| **zonotope** (Z) | e | e/o | e | e/o | e/o | e | e | o | o | o | o |
| **polytope** (P) | e | e/o | e | e/o | e/o | e | e | o | o | o | o |
| **conZonotope** (cZ) | e | e/o | e | e/o | e/o | e | e | o | o | o | o |
| **zonoBundle** (zB) | e | e/o | e | e/o | e/o | e | e | o | o | o | o |
| **ellipsoid** (E) | e | e | e | e | e | e | o | o | o | o | o |
| **capsule** (C) | e | e | e | e | e | o | e | o | o | o | o |
| **polyZonotope** (pZ) | o | o | o | o | o | o | o | o | o | o | o |
| **conPolyZono** (cPZ) | o | o | o | o | o | o | o | o | o | o | o |
| **taylm** | o | o | o | o | o | o | o | o | o | o | o |
| **spectraShadow** (SpS) | e | e | e | e | e | o | o | o | o | o | o |

#### 2.1.2.2 isIntersecting

The method `isIntersecting` checks if two sets intersect. Given two sets $\mathcal{S}_1, \mathcal{S}_2 \subset \mathbb{R}^n$, the method `isIntersecting` is defined as

$$\texttt{isIntersecting}(\mathcal{S}_1, \mathcal{S}_2) = \begin{cases} \texttt{true}, & \mathcal{S}_1 \cap \mathcal{S}_2 \neq \emptyset, \\ \texttt{false} & \text{otherwise.} \end{cases}$$

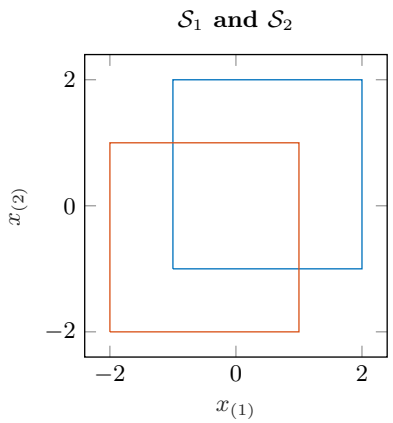
Since intersection checks can be computationally expensive, we implemented over-approximative algorithms for some set representations (see Tab. 3). If the over-approximative algorithm returns `false`, it is guaranteed that the sets do not intersect. However, if the over-approximative algorithm returns `true`, the sets could possibly not intersect. To execute the over-approximative instead of the exact algorithm, one has to add the flag 'approx':

```
res = isIntersecting(S1,S2,'approx');
```

Let us demonstrate the method `isIntersecting` by an example:

```
% sets S1 and S2
S1 = interval([-1;-1],[2;2]);
S2 = interval([-2;-2],[1;1]);

% intersection check
res = isIntersecting(S1,S2)
```

```
Command Window:

res = true
```




#### 2.1.2.3 isFullDim

The method `isFullDim` checks if a set is full-dimensional, that is, if the dimension of its affine hull is equal to the dimension of its ambient space. Given a set $\mathcal{S} \subset \mathbb{R}^n$, the method `isFullDim`

Table 3: Intersection checks implemented by the function `isIntersecting`$(\mathcal{S}_1, \mathcal{S}_2)$ in CORA. The shortcuts e (exact check) and o (over-approximation) are used. If both, an exact and an over-approximative algorithm are implemented, we write e/o.

| | **I** | **Z** | **P** | **cZ** | **zB** | **E** | **C** | **tay** | **pZ** | **cPZ** | **hs** | **cHp** | **ls** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **interval** (I) | e | e/o | e/o | e/o | e/o | o | o | o | o | o | e | e/o | o |
| **zonotope** (Z) | e/o | e/o | e/o | e/o | e/o | o | o | o | o | o | e | e/o | o |
| **polytope** (P) | e/o | e/o | e | e/o | e/o | o | o | o | o | o | e | e/o | o |
| **conZonotope** (cZ) | e/o | e/o | e/o | e/o | e/o | o | o | o | o | o | e | e/o | o |
| **zonoBundle** (zB) | e/o | e/o | e/o | e/o | e/o | o | o | o | o | o | e | e/o | o |
| **ellipsoid** (E) | o | o | o | o | o | e | o | o | o | o | e | o | o |
| **capsule** (C) | o | o | o | o | o | o | e | o | o | o | e | o | o |
| **taylm** (tay) | o | o | o | o | o | | | | o | o | o | o | o |
| **polyZonotope** (pZ) | o | o | o | o | o | | | o | o | o | o | o | o |
| **conPolyZono** (cPZ) | o | o | o | o | o | o | o | o | o | o | o | o | o |
| **levelSet** (ls) | o | o | o | o | o | o | o | o | o | o | | | |

is defined as

$$\texttt{isFullDim}(\mathcal{S}) = \begin{cases} \texttt{true}, & \exists x \in \mathcal{S}, \epsilon > 0 : \; x + \epsilon\mathcal{B} \subseteq \mathcal{S}, \\ \texttt{false} & \text{otherwise}, \end{cases}$$

where $\mathcal{B} = \{x \mid ||x||_2 \leq 1\} \subset \mathbb{R}^n$ is the unit ball. Let us demonstrate the method `isFullDim` by an example:

```
% sets S1 and S2
S1 = zonotope([1 2 1;3 1 2]);
S2 = zonotope([1 2 1;3 4 2]);

% check if full-dimensional
res = isFullDim(S1)
res = isFullDim(S2)
```

```
Command Window:

res = true

res = false
```

#### 2.1.2.4 isequal

The method `isequal` checks if two sets are identical. Optionally, a tolerance can be set to reduce the effect of floating-point deviations. Given two sets $\mathcal{S}_1, \mathcal{S}_2 \subset \mathbb{R}^n$, the method `isequal` is defined as

$$\texttt{isequal}(\mathcal{S}_1, \mathcal{S}_2, \texttt{tol}) = \begin{cases} \texttt{true}, & \mathcal{S}_1 = \mathcal{S}_2 \\ \texttt{false} & \text{otherwise}. \end{cases}$$

Let us demonstrate the method `isequal` by an example:

```
% sets S1 and S2
S1 = zonotope([0 1 1 0; ...
               0 1 0 1]);
S2 = zonotope([0 1 1 0; ...
               0 1 0 1]);

% equality check
res = isequal(S1,S2)
```

```
Command Window:

res = true
```

#### 2.1.2.5 representsa

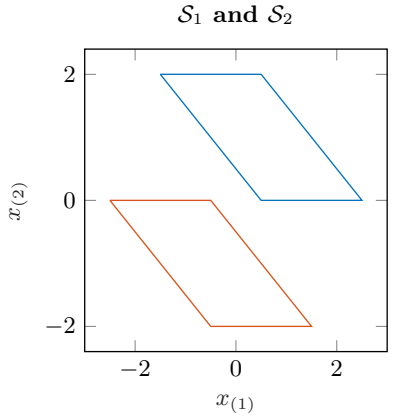
The method `representsa` checks if a set can equivalently be represented by a different set, e.g. a special case. Given a set $\mathcal{S} \subset \mathbb{R}^n$ and a string `type`, the method `representsa` is defined as

$$\texttt{representsa}(\mathcal{S}, \texttt{type}, \texttt{tol}) = \begin{cases} \texttt{true}, & \mathcal{S} \text{ can be represented by } \texttt{type}, \\ \texttt{false} & \text{otherwise}, \end{cases}$$

where `type` is the class name of another set representation or a special case, e.g. `'point'`. Let us demonstrate the method `representsa` by an example to check if a given set is empty:

```
% set S (intersection)
S1 = polytope(...
     [-1 -1;0 -1;0 1;1 1], ...
     [-0.5; 0; 2; 2.5]);

S2 = polytope(...
     [-1 -1;0 -1;0 1;1 1], ...
     [2.5; 2; 0; -0.5]);

S = S1 & S2;

% check if set is empty
res = representsa(S,'emptySet')
```

```
Command Window:

res = true
```




Note: This function replaces the function `isempty`. The main reason is that `isempty` is also called implicitly by MATLAB in various circumstances. For example, `isempty` is called on each workspace variable at a breakpoint, which can lead to long loading times if the empty check is expensive for a given set representation.

### 2.1.3 Set Properties

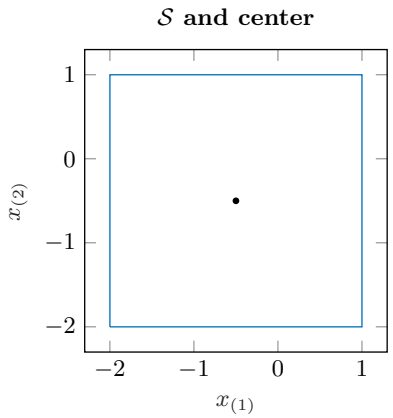
In this subsection, we describe the methods that compute geometric properties of sets.

#### 2.1.3.1 center

The method `center` returns the center of a set. Let us demonstrate the method `center` by an example:

```
% set S
S = interval([-2;-2],[1;1]);

% compute center
res = center(S)
```

```
Command Window:

res =

   -0.5000
   -0.5000
```



#### 2.1.3.2 dim

The method `dim` returns the dimension of the ambient space of a set, that is, the dimension in which a set is defined. Let us demonstrate the method `dim` by an example:

```
% set S
S = zonotope([0 1 0 2; ...
              3 1 1 0; ...
              1 1 0 1]);

% dimension of the set
res = dim(S)
```

```
Command Window:

res = 3
```

#### 2.1.3.3 norm

The method `norm` returns the maximum norm value of the vector norm for points inside a set $\mathcal{S} \subset \mathbb{R}^n$:

$$\texttt{norm}(\mathcal{S}, p) = \max_{x \in \mathcal{S}} \|x\|_p, \quad p \in \{1, 2, \ldots, \infty\},$$

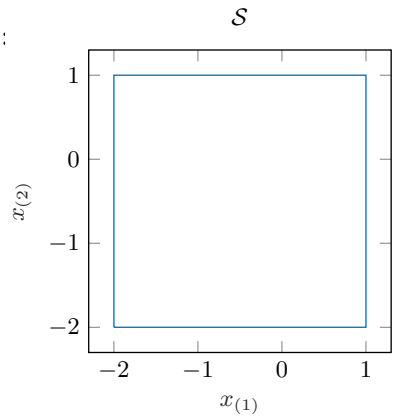
where the $p$-norm $\|\cdot\|_p$ is defined as

$$\|x\|_p = \left( \sum_{i=1}^{n} |x_i|^p \right)^{1/p}.$$

Let us demonstrate the method `norm` by an example:

```
% set S
S = zonotope([-0.5 1.5 0; ...
              -0.5 0 1.5]);

% norm of the set
res = norm(S,2)
```

```
Command Window:

res =

2.8284
```




#### 2.1.3.4 vertices

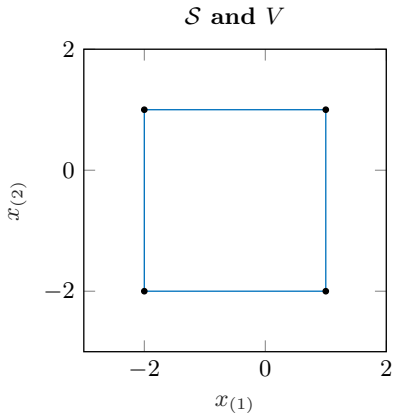
Given a set $\mathcal{S} \subset \mathbb{R}^n$, the method `vertices` computes the vertices $v_1, \ldots, v_q$, $v_i \in \mathbb{R}^n$ of the set:

$$[v_1, \ldots, v_q] = \texttt{vertices}(\mathcal{S}).$$

Please note that the computation of vertices can be computationally demanding for complex-shaped and/or high-dimensional sets. Let us demonstrate the method `vertices` by an example:

```
% set S
S = interval([-2;-2], ...
             [1;1]);

% compute vertices
V = vertices(S)
```

```
Command Window:

V =

     1     1    -2    -2
     1    -2     1    -2
```

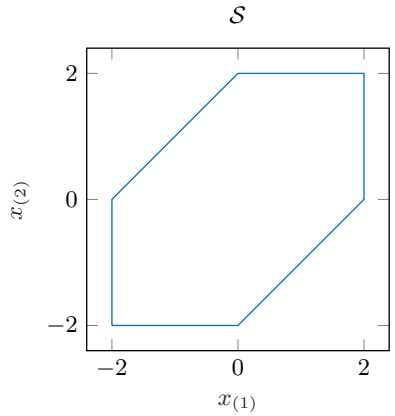



#### 2.1.3.5 volume

The method `volume` returns the volume of a set. Let us demonstrate the method `volume` by an example:

```
% set S
S = zonotope([0 1 1 0; ...
              0 1 0 1]);

% volume of the set
res = volume(S)
```

```
Command Window:

res = 12
```




### 2.1.4 Auxiliary Operations

In this subsection, we describe useful auxiliary operations.

#### 2.1.4.1 cubMap

The method `cubMap` implements the cubic map of a set. Given a set $\mathcal{S} \subset \mathbb{R}^n$, the cubic map is defined as

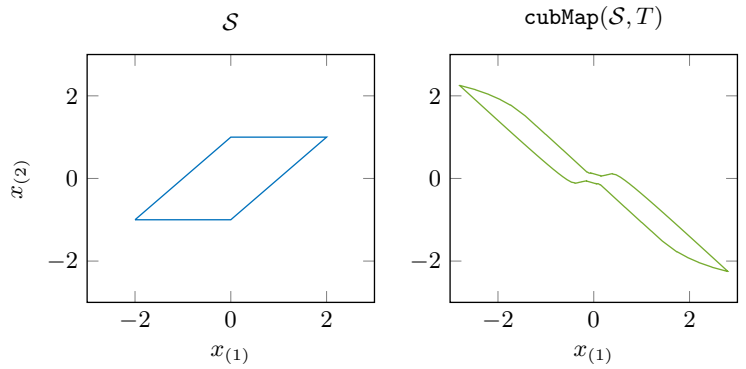
$$\texttt{cubMap}(\mathcal{S}, Q) = \left\{ x \;\middle|\; x_{(i)} = \sum_{j=1}^{n} s_{(j)} \; (s^T \; T_{i,j} \; s), \; s \in \mathcal{S}, \; i = 1 \dots w \right\}, \; T_{i,j} \in \mathbb{R}^{n \times n},$$

where $x_{(i)}$ is the $i$-th value of the vector $x$. If the corresponding set representation is not closed under cubic maps, `cubMap` returns an over-approximation. If `cubMap` is called with three different sets $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3 \subset \mathbb{R}^n$ as input arguments, the method computes the mixed cubic map:

$$\texttt{cubMap}(\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, Q) = \left\{ x \;\middle|\; x_{(i)} = \sum_{j=1}^{n} s_{1(j)} \; (s_2^T \; T_{i,j} \; s_3), \; s_1 \in \mathcal{S}_1, \; s_2 \in \mathcal{S}_2, \; s_3 \in \mathcal{S}_3, \right.$$
$$\left. i = 1 \dots w \right\}, \; T_{i,j} \in \mathbb{R}^{n \times n},$$

Let us demonstrate the method `cubMap` by an example:

```
% set and matrices
S = polyZonotope([0;0], ...
                 [1 1;1 0], ..
                 [],eye(2));

T{1,1} = 0.4*[1 2; -1 2];
T{1,2} = 0.4*[-3 0; 1 1];
T{2,1} = 0.05*[2 0; -2 1];
T{2,2} = 0.05*[-3 0; -21 -1];

% cubic map
res = cubMap(S,T);
```




#### 2.1.4.2 enclose

The method `enclose` computes an enclosure of a set and its linear transformation. Given the sets $\mathcal{S}_1, \mathcal{S}_2 \in \mathbb{R}^n$ and the matrix $M \in \mathbb{R}^{n \times n}$, `enclose` computes the set

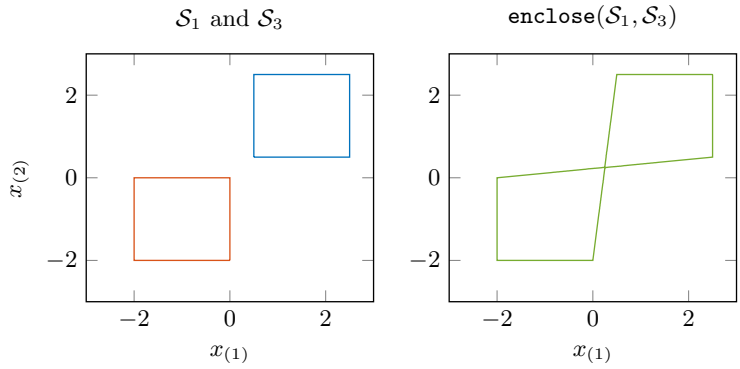
$$\texttt{enclose}(\mathcal{S}_1, M, \mathcal{S}_2) = \{\lambda s_1 + (1-\lambda)(M s_1 + s_2) \mid s_1 \in \mathcal{S}_1,\ s_2 \in \mathcal{S}_2,\ \lambda \in [0,1]\}\,. \tag{1}$$

If the set as defined in (1) cannot be computed exactly for the corresponding set representation, `enclose` returns an over-approximation. For convenience, the method can also be called with only two input arguments:

$$\texttt{enclose}(\mathcal{S}_1, \mathcal{S}_3) = \texttt{enclose}(\mathcal{S}_1, M, \mathcal{S}_2), \quad \mathcal{S}_3 = (M \otimes \mathcal{S}_1) \oplus \mathcal{S}_2.$$

Let us demonstrate the method `enclose` by an example:

```
% sets S1,S2 and matrix M
S1 = polyZonotope([1.5;1.5], ...
                  [1 0;0 1], ...
                  [],eye(2));
S2 = [0.5;0.5];
M = [-1 0;0 -1];

% apply method enclose
S3 = M*S1 + S2;

res = enclose(S1,M,S2);
res = enclose(S1,S3);
```



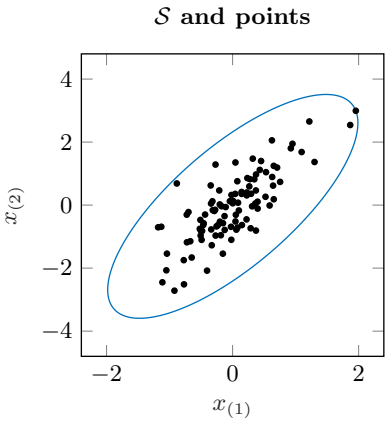

#### 2.1.4.3 enclosePoints

Given a point cloud $P = [p_1, \ldots, p_m]$, $p_i \in \mathbb{R}^n$, the static method `enclosePoints` computes a set $\mathcal{S} \subset \mathbb{R}^n$ that tightly encloses the point cloud:

$$\mathcal{S} = \texttt{enclosePoints}\big([p_1, \ldots, p_m]\big), \quad \forall i = 1, \ldots, m:\ p_i \in \mathcal{S}.$$

Let us demonstrate the method `enclosePoints` by an example:

```
% random point cloud
mu = [0 0];
sigma = [0.3 0.4; 0.4 1];
points = mvnrnd(mu,sigma,100)';

% compute enclosing set
S = ellipsoid.enclosePoints(points);
```




#### 2.1.4.4 enlarge

Given a set $\mathcal{S} \subset \mathbb{R}^n$ and a scaling factor $\lambda \in \mathbb{R}_+$, the method `enlarge` is defined as

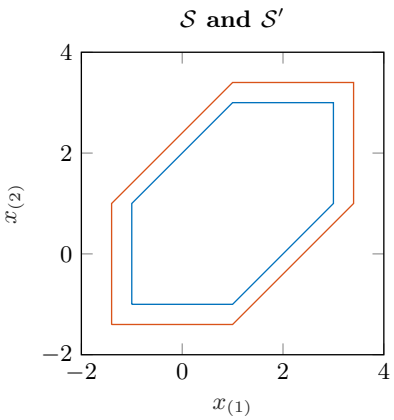
$$\mathcal{S}' = \texttt{enlarge}\,(\mathcal{S}, \lambda) = \texttt{center}(\mathcal{S}) + \lambda(\mathcal{S} - \texttt{center}(\mathcal{S})),$$

such that

$$\begin{aligned} \mathcal{S} &\subseteq \mathcal{S}' \quad \text{for } \lambda \geq 1, \\ \mathcal{S} &\supseteq \mathcal{S}' \quad \text{otherwise.} \end{aligned}$$

Let us demonstrate the method `enlarge` by an example:

```
% set S and scaling lambda
S = zonotope([1;1],[1 0 1; 0 1 1]);
lambda = 1.2;

% enlarge set S
S_ = enlarge(S,lambda);
```




#### 2.1.4.5 generateRandom

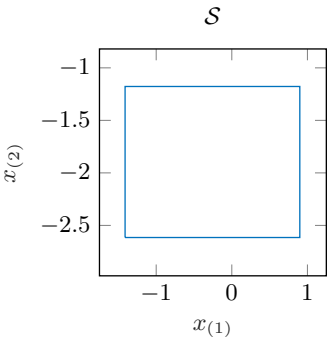
The static method `generateRandom` randomly generates a set of the given set representation. If no input arguments are provided, the method generates a random set of arbitrary dimension. The desired dimension and other specifications for the set can be provided by name-value pairs:

$$\mathcal{S} = \texttt{generateRandom}(\texttt{'Dimension'}, n), \quad \mathcal{S} \subset \mathbb{R}^{\texttt{n}}.$$

The additionally supported name-value pairs of the method `generateRandom` for each class are detailed in their respective function description. Let us demonstrate the method `generateRandom` by an example:

```
% generate random set
S = interval.generateRandom( ...
      'Dimension',2);
```

```
Command Window:

S =
  [-1.4113, 0.8993]
  [-2.6153, -1.1773]
```



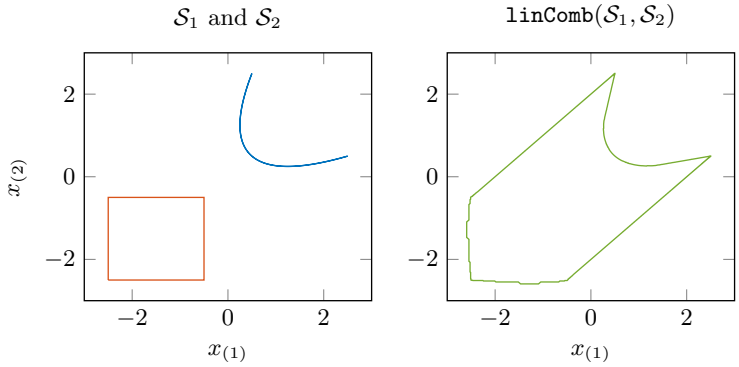

#### 2.1.4.6 linComb

The method `linComb` implements the linear combination of two sets. Given two sets $\mathcal{S}_1, \mathcal{S}_2 \subset \mathbb{R}^n$, their linear combination is defined as

$$\texttt{linComb}(\mathcal{S}_1, \mathcal{S}_2) = \{\lambda s_1 + (1-\lambda) s_2 \mid s_1 \in \mathcal{S}_1,\ s_2 \in \mathcal{S}_2,\ \lambda \in [0,1]\}\,.$$

Note that for convex sets the linear combination is identical to the convex hull (see Sec. 2.1.1.4). For non-convex sets, however, the two operations differ. Let us demonstrate the method `linComb` by an example:

```
% set S1 and S2
S1 = polyZonotope([0.5;0.5],...
                  [1 1;-1 1],...
                  [],[1 2]);
S2 = zonotope([-1.5;-1.5],...
              [1 0;0 1]);

% linear combination
res = linComb(S1,S2);
```




#### 2.1.4.7 randPoint

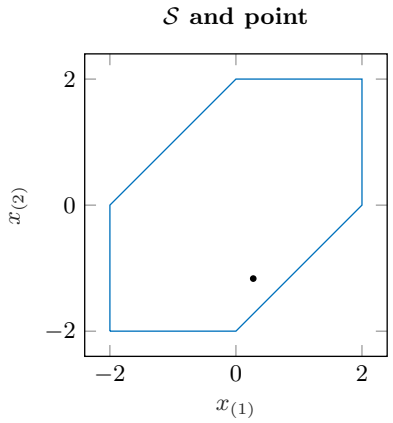
The method `randPoint` returns random points located inside a set. Given a set $\mathcal{S} \subset \mathbb{R}^n$, the method `randPoint` generates random points $p = [p_1, \ldots, p_N] \in \mathbb{R}^{n \times N}$ with $p_1, \ldots, p_N \in \mathcal{S}$:

$$\begin{aligned} p &= \texttt{randPoint}(\mathcal{S}), \\ p &= \texttt{randPoint}(\mathcal{S}, N), \\ p &= \texttt{randPoint}(\mathcal{S}, N, \texttt{type}), \end{aligned}$$

where $N \in \mathbb{N}_{>0}$ is the desired number of points, and `type` specifies the desired type random points. The setting `type = 'extreme'` aims to generate points close to or on the boundary of the set, while `type = 'standard'` generates arbitrary points within the set. In contrast, the setting `type = 'uniform'` generates uniformly distributed points within the set, potentially at the cost of a longer runtime. The default values are $N = 1$ and `type = 'standard'`. Let us demonstrate the method `randPoint` by an example:

```
% set S
S = zonotope([0 1 1 0; ...
              0 1 0 1]);

% random point
p = randPoint(S)
```

```
Command Window:

p =

    0.2747
   -1.1657
```



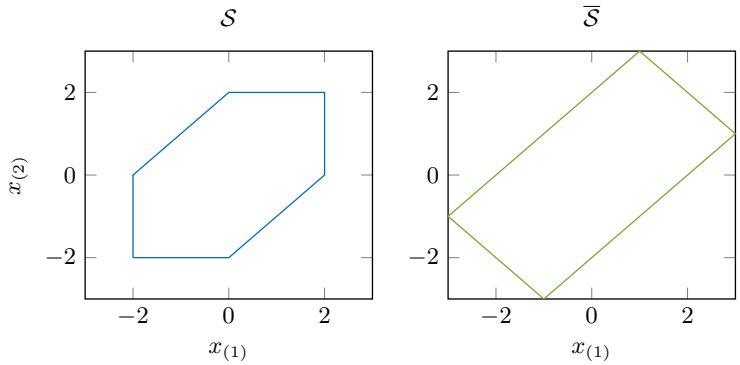

#### 2.1.4.8 reduce

The method `reduce` encloses a set by another set with a smaller representation size. Given a set $\mathcal{S} \subset \mathbb{R}^n$, the method `reduce` computes

$$\texttt{reduce}(\mathcal{S}, \texttt{method}, \texttt{order}) = \overline{\mathcal{S}}, \quad \mathcal{S} \subseteq \overline{\mathcal{S}}, \tag{2}$$

where the representation size of $\overline{\mathcal{S}}$ is smaller than the one of $\mathcal{S}$. The parameter `method` in (2) is a string that specifies the algorithm to be applied, see Tab. 4. The parameter `order` in (2) is a measure for the desired representation size of the resulting set $\overline{\mathcal{S}}$. Currently, the method `reduce` is implemented for the zonotopic set representations `zonotope` (see Sec. 2.2.1.1), `conZonotope` (see Sec. 2.2.1.9), `polyZonotope` (see Sec. 2.2.1.5), and `probZonotope` (see Sec. 2.2.1.11), where `order` $= \frac{p}{n}$ is defined as the division of the number of generator vectors $p$ by the system dimension $n$. Let us demonstrate the method `reduce` by an example:

```
% set S
S = zonotope([0 1 1 0; ...
              0 1 0 1]);

% reduce rep. size
S_ = reduce(S,'pca',1);
```




#### 2.1.4.9 supportFunc

The method `supportFunc` computes the support function for a specific direction. Given a set $\mathcal{S} \in \mathbb{R}^n$ and a vector $l \in \mathbb{R}^n$, the support function is defined as

$$\texttt{supportFunc}(\mathcal{S}, l) = \max_{x \in \mathcal{S}} \; l^T \; x.$$

The function also supports the computation of the lower bound, which can be calculated using the flag `'lower'`:

$$\texttt{supportFunc}(\mathcal{S}, l, \texttt{'lower'}) = \min_{x \in \mathcal{S}} \; l^T \; x.$$

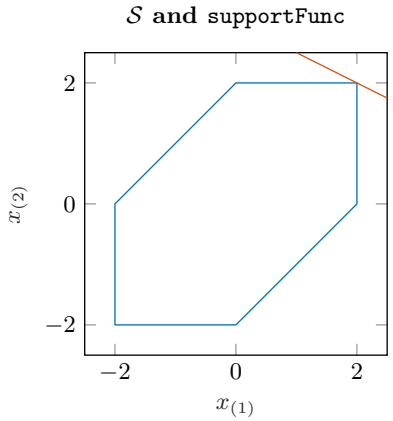
Table 4: Reduction techniques for zonotopic set representations.

| Technique | Primary use | Reference |
|---|---|---|
| `cluster` | Reduction to low order by clustering generators | [30, Sec. III.B] |
| `combastel` | Reduction of high to medium order | [31, Sec. 3.2] |
| `constOpt` | Reduction to low order by optimization | [30, Sec. III.D] |
| `girard` | Reduction of high to medium order | [32, Sec. .4] |
| `methA` | Reduction to low order by volume minimization (A) | Meth. A, [33, Sec. 2.5.5] |
| `methB` | Reduction to low order by volume minimization (B) | Meth. B, [33, Sec. 2.5.5] |
| `methC` | Reduction to low order by volume minimization (C) | Meth. C, [33, Sec. 2.5.5] |
| `scott` | Reduction to low order | [34, Appendix] |
| `pca` | Reduction of high to medium order using PCA | [30, Sec. III.A] |

Additionally, one can return both the lower and upper bounds by using the flag `'range'`. Let us demonstrate the method `supportFunc` by an example:

```
% set S and vector l
S = zonotope([0 1 1 0; ...
              0 1 0 1]);
l = [1;2];

% compute support function
res = supportFunc(S,l)
```

```
Command Window:

res = 6
```




#### 2.1.4.10 plot

The method `plot` visualizes a 2-dimensional projection of the boundary of a set. Given a set $\mathcal{S} \subset \mathbb{R}^n$, the method `plot` supports the following syntax:

$$\begin{aligned}
&\texttt{han} = \texttt{plot}(\mathcal{S}),\\
&\texttt{han} = \texttt{plot}(\mathcal{S}, \texttt{dims}),\\
&\texttt{han} = \texttt{plot}(\mathcal{S}, \texttt{dims}, \texttt{linespec}),\\
&\texttt{han} = \texttt{plot}(\mathcal{S}, \texttt{dims}, \texttt{namevaluepairs}),
\end{aligned}$$

where `han` is a handle to the plotted MATLAB graphics object and the additional input arguments are defined as

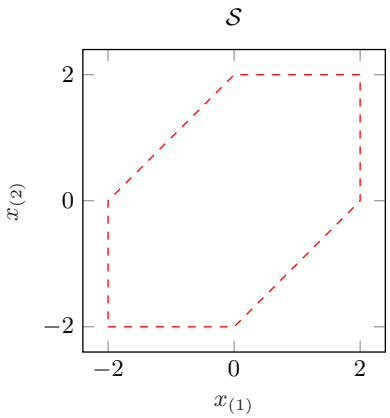
- `dims`: Integer vector $\texttt{dims} \in \mathbb{N}^2_{\leq n}$ specifying the dimensions for which the projection is visualized (default value: `dim = [1 2]`).
- `linespec`: (optional) line specifications, e.g., `'--*r'`, as supported by MATLAB[12].
- `namevaluepairs`: (optional) further specifications as name-value pairs, e.g., `'LineWidth',2` and `'FaceColor',[.5 .5 .5]`, as supported by MATLAB. If the plot is not filled, these

[12] `https://de.mathworks.com/help/matlab/ref/linespec.html`

are the built-in Line Properties[13], if the plot is filled, they correspond to the Patch Properties[14].

Let us demonstrate the method `plot` by an example:

```
% set S
S = zonotope([0 1 1 0; ...
              0 1 2 1; ...
              0 1 0 1]);

% visualization
plot(S,[1,3],'--r');
```




#### 2.1.4.11 project

The method `project` projects a set to a lower-dimensional, axis-aligned subspace. Given a set $\mathcal{S} \subset \mathbb{R}^n$ and a vector of subspace indices $\texttt{dims} \in \mathbb{N}^m_{\leq n}$, the method `project` returns

$$\texttt{project}(\mathcal{S}, \texttt{dims}) = \left\{ [s_{(\texttt{dims}_{(1)})}, \ldots, s_{(\texttt{dims}_{(m)})}]^T \;\middle|\; s \in \mathcal{S} \right\} \subset \mathbb{R}^m,$$

where $s_{(i)}$ denotes the i-th entry of vector $s$. Let us demonstrate the method `project` by an example:

```
% set S
S = interval([1;2;5;0], ...
             [3;3;7;2]);

% projection
res = project(S,[1 3 4]);
```

```
Command Window:

res =
[1.00000,3.00000]
[5.00000,7.00000]
[0.00000,2.00000]
```

A set can be projected into a higher-dimensional space using the function `projectHighDim`, where the new dimensions are bounded at 0. On the other hand, the function `lift` lifts the set into a higher-dimensional space with unbounded new dimensions.

[13] https://de.mathworks.com/help/matlab/ref/matlab.graphics.chart.primitive.line-properties.html

[14] https://de.mathworks.com/help/matlab/ref/matlab.graphics.primitive.patch-properties.html

## 2.2 Set Representations

The basis of any efficient reachability analysis is an appropriate set representation. On the one hand, the set representation should be general enough to describe the reachable sets accurately; on the other hand, it is crucial that the set representation facilitates efficient and scalable operations on them. CORA provides a wide range of set representations that are explained in detail in this section. Tab. 5 shows the supported conversions between set representations. In order to convert a set, it is sufficient to pass the current set to the class constructor of the target set representation, as demonstrated by the following example:

```
% create zonotope object
Z = zonotope([1 2 1;0 1 -1]);

% convert to other set representations
I = interval(Z);           % over-approximative conversion to an interval
P = polytope(Z);       % exact conversion to polytope
```

Table 5: Set conversions supported by CORA. The row headers represent the original set representation and the column headers the target set representation after conversion. The shortcuts e (exact conversion) and o (over-approximation) are used.

| | **Z** | **zB** | **pZ** | **cPZ** | **cZ** | **P** | **I** | **tay** | **C** | **E** | **SpS** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **zonotope** (Z, Sec. 2.2.1.1) | - | e | e | e | e | e | o | e | o | o | e |
| **zonoBundle** (zB, Sec. 2.2.1.8) | o | - | e | e | e | e | o | | | | e |
| **polyZonotope** (pZ, Sec. 2.2.1.5) | o | | - | e | | o | o | o | | o | o |
| **conPolyZono** (cPZ, Sec. 2.2.1.6) | o | o | o | - | o | o | o | o | o | o | |
| **probZonotope** (probZ, Sec. 2.2.1.11) | o | | | | | | | | | | |
| **conZonotope** (cZ, Sec. 2.2.1.9) | o | e | e | e | - | e | o | | | | e |
| **polytope** (P, Sec. 2.2.1.4) | | e | e | e | e | - | o | | | | e |
| **interval** (I, Sec. 2.2.1.2) | e | e | e | e | e | e | - | | o | o | e |
| **taylm** (tay, Sec. 2.2.3.1) | | | | e | | | o | - | | | |
| **capsule** (C, Sec. 2.2.1.7) | o | | | e | | | o | | - | | e |
| **ellipsoid** (E, Sec. 2.2.1.3) | o | | | e | | | o | | | - | e |
| **spectraShadow** (SpS, Sec. 2.2.1.10) | o | o | o | o | o | | o | | | | - |

### 2.2.1 Basic Set Representations

We first introduce basic set representations predominantly used to represent reachable sets.

#### 2.2.1.1 Zonotopes

A zonotope $\mathcal{Z} \subset \mathbb{R}^n$ is defined as

$$\mathcal{Z} := \left\{ c + \sum_{i=1}^{p} \beta_i g^{(i)} \;\middle|\; \beta_i \in [-1,1] \right\}, \tag{3}$$

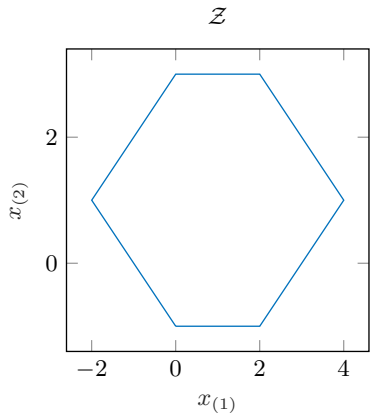
where $c \in \mathbb{R}^n$ is the center and $g^{(i)} \in \mathbb{R}^n$ are the generators. The zonotope order $\rho$ is defined as $\rho = \frac{p}{n}$ and represents a dimensionless measure for the representation size.

Zonotopes are represented in CORA by the class `zonotope`. An object of class `zonotope` can be constructed as follows:

$$\mathcal{Z} = \texttt{zonotope}(c, G),$$
$$\mathcal{Z} = \texttt{zonotope}(Z),$$

where $G = [g^{(1)}, \dots, g^{(p)}]$, $Z = [c, G]$, and $c, g^{(i)}$ are defined as in (3). Let us demonstrate the construction of a zonotope by an example:

```
% construct zonotope
c = [1;1];
G = [1 1 1; 1 -1 0];

zono = zonotope(c,G);
```




A more detailed example for zonotopes is provided in Sec. 10.1.1 and in the file *examples/contSet/example_zonotope.m* in the CORA toolbox.

A zonotope can be interpreted as the Minkowski addition of line segments $l^{(i)} = [-1, 1]g^{(i)}$. The step-by-step construction of a two-dimesional zonotope is visualized in Fig. 2. Zonotopes are a compact representation of sets in high-dimensional space. More importantly, operations required for reachability analysis, such as linear maps (see Sec. 2.1.1.1) and Minkowski addition (see Sec. 2.1.1.2) can be computed efficiently and exactly, and others, such as convex hull computation (see Sec. 2.1.1.4) can be tightly over-approximated [32].

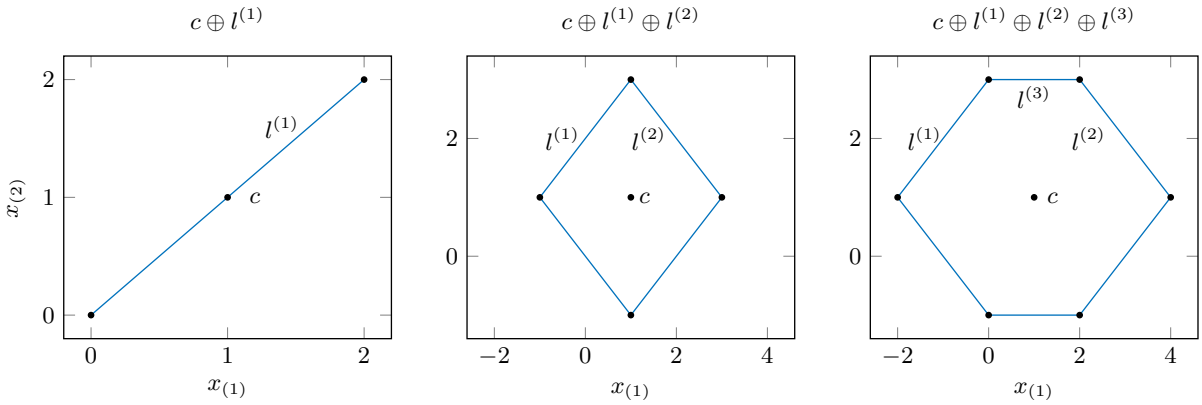



Figure 2: Step-by-step construction of a zonotope.

In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5), the class `zonotope` supports additional methods which are listed in Appendix A.1.

#### 2.2.1.2 Intervals

A real-valued multi-dimensional interval

$$\mathcal{I} := \{x \in \mathbb{R}^n \mid \underline{x}_i \leq x_i \leq \overline{x}_i \; \forall i = 1, \ldots, n\} \tag{4}$$

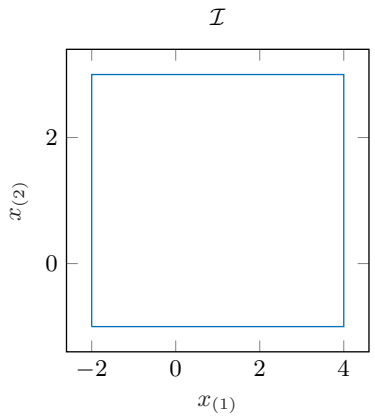
is a connected subset of $\mathbb{R}^n$ and can be specified by a lower bound $\underline{x} \in \mathbb{R}^n$ and upper bound $\overline{x} \in \mathbb{R}^n$.

Intervals are represented in CORA by the class `interval`. An object of class `interval` can be constructed as follows:

$$\mathcal{I} = \texttt{interval}(\underline{x}, \overline{x})$$

where $\underline{x}, \overline{x}$ are defined as in (4). A detailed description of how intervals are treated in CORA can be found in [11]. Let us demonstrate the construction of an interval by an example:

```
% construct interval
lb = [-2; -1];
ub = [4; 3];

int = interval(lb,ub);
```




A more detailed example for intervals is provided in Sec. 10.1.2 and in the file *examples/contSet/example_interval.m* in the CORA toolbox. Intervals can also be used for range bounding as it described in Sec. 2.2.3. In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5), the class `interval` supports additional methods, which are listed in Appendix A.2.

#### 2.2.1.3 Ellipsoids

An ellipsoid is a geometric object in $\mathbb{R}^n$. Ellipsoids are parameterized by a center $q \in \mathbb{R}^n$ and a positive semi-definite, symmetric shape matrix $Q \in \mathbb{R}^{n \times n}$ and defined as[15]

$$\mathcal{E} := \left\{ x \in \mathbb{R}^n \;\middle|\; l^T x \leq l^T q + \sqrt{l^T Q l},\ \forall l \in \mathbb{R}^n \right\}. \tag{5}$$

If we assume $Q$ to be invertible (which holds true for non-degenerate ellipsoids), it can be equivalently defined as (see [35, Definition 2.1.3])

$$\mathcal{E} := \left\{ x \in \mathbb{R}^n \;\middle|\; (x-q)^T Q^{-1} (x-q) \leq 1 \right\}.$$

Ellipsoids have a compact representation increasing only with dimension. Linear maps (see Sec. 2.1.1.1) can be computed exactly and efficiently, Minkowski sum (see Sec. 2.1.1.2) and others can be tightly over-approximated.

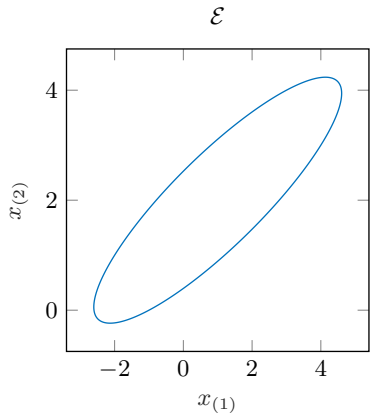
Ellipsoids are represented in CORA by the class `ellipsoid`. An object of class `ellipsoid` can be constructed as follows:

$$\mathcal{E} = \texttt{ellipsoid}(Q),$$
$$\mathcal{E} = \texttt{ellipsoid}(Q, q),$$

where $Q, q$ are defined as in (5). Let us demonstrate the construction of an ellipsoid by an example:

[15] https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-46.pdf, Sec. 2.2.4

```
% construct ellipsoid
Q = [13 7; 7 5];
q = [1; 2];

E = ellipsoid(Q,q);
```




A more detailed example for ellipsoids is provided in Sec. 10.1.3 and in the file *examples/contSet/example_ellipsoid.m* in the CORA toolbox. In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5), the class `ellipsoid` supports additional methods, which are listed in Appendix A.3.

#### 2.2.1.4 Polytopes

There exist two representations for polytopes: The halfspace representation (H-representation) and the vertex representation (V-representation). Details of the implemented functionalities can be found in [8].

**H-Representation of a Polytope**

The halfspace representation specifies a convex polytope $\mathcal{P}$ by the intersection of $q$ halfspaces $\mathcal{H}^{(i)}$: $\mathcal{P} = \mathcal{H}^{(1)} \cap \mathcal{H}^{(i)} \cap \ldots \cap \mathcal{H}^{(q)}$. A halfspace is one of the two parts obtained by bisecting the $n$-dimensional Euclidean space with a hyperplane $\mathcal{S} := \{x \,|\, a^T x = b\}, a \in \mathbb{R}^n, b \in \mathbb{R}$. The vector $a$ is the normal vector of the hyperplane and $b$ is the scalar product of any point on the hyperplane with the normal vector. From this follows that the corresponding halfspace is $\mathcal{H} := \{x \,|\, a^T x \leq b\}$. As the convex polytope $\mathcal{P}$ is the non-empty intersection of $q$ halfspaces, all $q$ inequalities have to be fulfilled simultaneously.

A convex polytope $\mathcal{P}$ is the bounded intersection of $q$ halfspaces:

$$\mathcal{P} := \Big\{ x \in \mathbb{R}^n \;\Big|\; A\,x \leq b \Big\}, \quad A \in \mathbb{R}^{q \times n}, b \in \mathbb{R}^q. \tag{6}$$

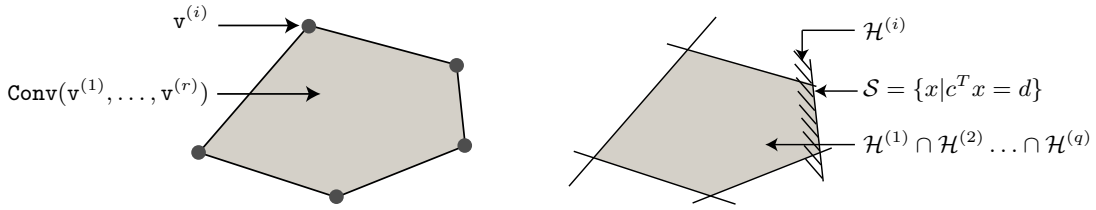
When the intersection is unbounded, one obtains a polyhedron [36].

**V-Representation of a Polytope**

A polytope with vertex representation is defined as the convex hull of a finite set of points in the $n$-dimensional Euclidean space. The points are also referred to as vertices and denoted by $\mathrm{v}^{(i)} \in \mathbb{R}^n$. A convex hull of a finite set of $r$ points $\mathrm{v}^{(i)} \in \mathbb{R}^n$ is obtained from their linear combination:

$$\mathtt{Conv}(\mathrm{v}^{(1)}, \ldots, \mathrm{v}^{(r)}) := \Big\{ \sum_{i=1}^{r} \alpha_i \mathrm{v}^{(i)} \;\Big|\; \alpha_i \in \mathbb{R},\, \alpha_i \geq 0,\, \sum_{i=1}^{r} \alpha_i = 1 \Big\}. \tag{7}$$

The halfspace and the vertex representation are illustrated in Fig. 3. Algorithms that convert from H- to V-representation and vice versa are presented in [37].




(a) $V-representation$ (b) $H-representation$

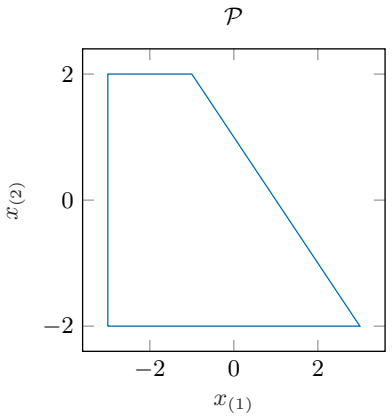
Figure 3: Possible representations of a polytope.

Polytopes are represented in CORA by the class `polytope`. An object of class `polytope` can be constructed as follows:

$$\begin{aligned}\mathcal{P} &= \mathtt{polytope}(V),\\ \mathcal{P} &= \mathtt{polytope}(A,b),\\ \mathcal{P} &= \mathtt{polytope}(A,b,Ae,be),\end{aligned}$$

where $V=[v^{(1)},\ldots,v^{(r)}]$, $v^{(i)}$ is defined as in (7), $A,b$ are defined as in (6), and $Ae,be$ are equality constraints that are used to compactly represent pairwise inequality constraints $a^\top x\leq b, a^\top x\geq b$. Let us demonstrate the construction of a polytope by an example:

```
% construct polytope (halfspace rep.)
A = [1 0 -1 0 1; 0 1 0 -1 1]';
b = [3; 2; 3; 2; 1];

poly = polytope(A,b);

% construct polytope (vertex rep.)
V = [-3 -3 -1 3; -2 2 2 -2];

poly = polytope(V);
```




A more detailed example for polytopes is provided in Sec. 10.1.4 and in the file *examples/contSet/example_polytope.m* in the CORA toolbox. In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5), the class `polytope` supports additional methods, which are listed in Appendix A.4.

#### 2.2.1.5 Polynomial Zonotopes

Polynomial zonotopes, which were first introduced in [38], are a non-convex set representation. In CORA we implemented the sparse representation of polynomial zonotopes described in [39]. A polynomial zonotope $\mathcal{PZ}\subset\mathbb{R}^n$ is defined as

$$\mathcal{PZ}:=\left\{c+\sum_{i=1}^{h}\left(\prod_{k=1}^{p}\alpha_k^{E_{(k,i)}}\right)G_{(\cdot,i)}+\sum_{j=1}^{q}\beta_j G_{I(\cdot,j)}\;\middle|\;\alpha_k,\beta_j\in[-1,1]\right\},\tag{8}$$

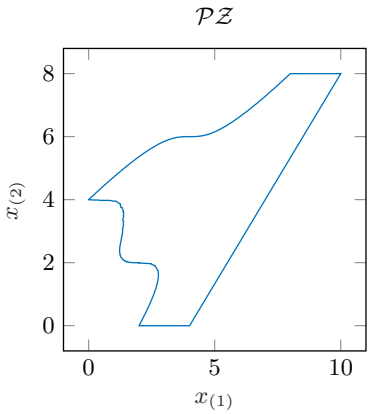
where $c\in\mathbb{R}^n$ is the center, $G\in\mathbb{R}^{n\times h}$ the matrix of dependent generators, $G_I\in\mathbb{R}^{n\times q}$ the matrix of independent generators, and $E\in\mathbb{N}_0^{p\times h}$ the exponent matrix. Since polynomial zonotopes can represent non-convex sets, and since they are closed under quadratic and higher-order maps, they are a good choice for reachability analysis.

Polynomial zonotopes are represented in CORA by the class `polyZonotope`. An object of class `polyZonotope` can be constructed as follows:

$$\mathcal{PZ} = \texttt{polyZonotope}(c, G, G_I, E),$$
$$\mathcal{PZ} = \texttt{polyZonotope}(c, G, G_I, E, id),$$

where $c, G, G_I, E$ are defined as in (8). The vector $id \in \mathbb{N}^p_{>0}$ stores unambiguous identifiers for the dependent factors $\alpha_k$, which is important for dependency preservation as described in [40]. Let us demonstrate the construction of a polynomial zonotope by an example:

```
% construct polynomial zonotope
c = [4;4];
G = [2 1 2; 0 2 2];
E = [1 0 3;0 1 1];
GI = [1;0];

pZ = polyZonotope(c,G,GI,E);
```




This example defines the polynomial zonotope

$$\mathcal{PZ} = \left\{ \begin{bmatrix}4\\4\end{bmatrix} + \alpha_1 \begin{bmatrix}2\\0\end{bmatrix} + \alpha_2 \begin{bmatrix}1\\2\end{bmatrix} + \alpha_1^3\alpha_2 \begin{bmatrix}2\\2\end{bmatrix} + \beta_1 \begin{bmatrix}1\\0\end{bmatrix} \;\middle|\; \alpha_1, \alpha_2, \beta_1 \in [-1, 1] \right\}.$$

The construction of this polynomial zonotope is visualized in Fig. 4: (a) shows the set spanned by the constant offset vector and the first and second dependent generator, (b) shows the addition of the dependent generator with the mixed term $\alpha_1^3\alpha_2$, (c) shows the addition of the independent generator, and (d) visualizes the final set.

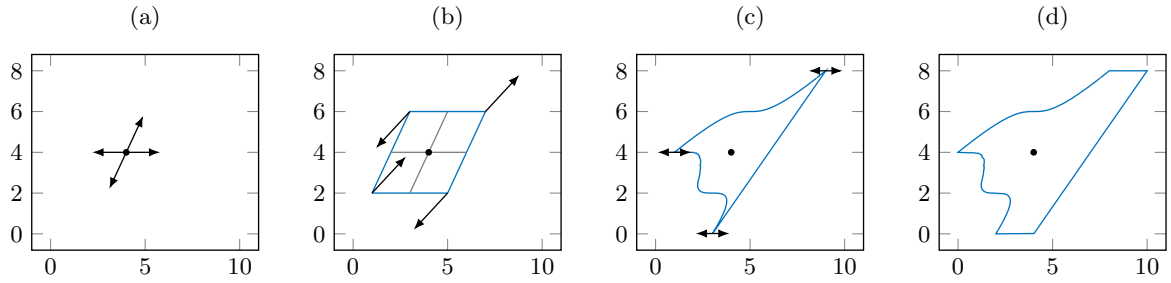



Figure 4: Step-by-step construction of a polynomial zonotope.

A more detailed example for polynomial zonotopes is provided in Sec. 10.1.5 and in the file *examples/contSet/example_polyZonotope.m* in the CORA toolbox.

#### 2.2.1.6 Constrained Polynomial Zonotopes

Constrained polynomial zonotopes as introduced in [41] extend the polynomial zonotopes in Sec. 2.2.1.5 by polynomial equality constraints on the dependent factors. Since constrained zonotopes are closed under all relevant set operations including intersection and union (see Tab. 1),

they are advantageous for many set-based algorithms. Furthermore, as shown in Tab. 5, most other set representations can be equivalently represented as constrained polynomial zonotopes, which further substantiates their importance. A constrained polynomial zonotope $\mathcal{CPZ} \subset \mathbb{R}^n$ is defined as

$$\begin{aligned}\mathcal{CPZ} := \Bigg\{ c + \sum_{i=1}^{h} \left( \prod_{k=1}^{p} \alpha_k^{E_{(k,i)}} \right) G_{(\cdot,i)} + \sum_{j=1}^{d} \beta_j G_{I(\cdot,j)} \;\Bigg|\; \\ \sum_{i=1}^{q} \left( \prod_{k=1}^{p} \alpha_k^{R_{(k,i)}} \right) A_{(\cdot,i)} = b,\ \alpha_k, \beta_j \in [-1,1] \Bigg\},\end{aligned} \tag{9}$$

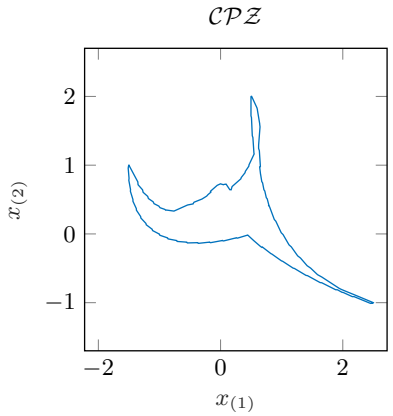
where $c \in \mathbb{R}^n$ is the constant offset, $G \in \mathbb{R}^{n \times h}$ the matrix of dependent generators, $G_I \in \mathbb{R}^{n \times d}$ the matrix of independent generators, $E \in \mathbb{N}_0^{p \times h}$ the exponent matrix, $A \in \mathbb{R}^{m \times q}$ is the matrix of constraint generators, $b \in \mathbb{R}^m$ is the constraint offset, and $R \in \mathbb{N}_0^{p \times q}$ is the constraint exponent matrix.

Constrained polynomial zonotopes are represented in CORA by the class `conPolyZono`. An object of class `conPolyZono` can be constructed as follows:

$$\begin{aligned}\mathcal{CPZ} &= \texttt{conPolyZono}(c, G, E), \\ \mathcal{CPZ} &= \texttt{conPolyZono}(c, G, E, G_I), \\ \mathcal{CPZ} &= \texttt{conPolyZono}(c, G, E, G_I, id), \\ \mathcal{CPZ} &= \texttt{conPolyZono}(c, G, E, A, b, R), \\ \mathcal{CPZ} &= \texttt{conPolyZono}(c, G, E, A, b, R, G_I), \\ \mathcal{CPZ} &= \texttt{conPolyZono}(c, G, E, A, b, R, G_I, id),\end{aligned}$$

where $c, G, G_I, E, A, b, R$ are defined as in (9). The vector $id \in \mathbb{N}_{>0}^p$ stores unambiguous identifiers for the dependent factors $\alpha_k$, which is important for dependency preservation as described in [40]. Let us demonstrate constrained polynomial zonotopes with an example:

```
% construct conPolyZono object
c = [0;0];
G = [1 0 1 -1;0 1 1 1];
E = [1 0 1 2;0 1 1 0;0 0 1 1];
A = [1 -0.5 0.5];
b = 0.5;
R = [0 1 2;1 0 0;0 1 0];

cPZ = conPolyZono(c,G,E,A,b,R);
```




This example defines the constrained polynomial zonotope

$$\begin{aligned}\mathcal{CPZ} = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \alpha_1 + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \alpha_2 + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \alpha_1\alpha_2\alpha_3 + \begin{bmatrix} -1 \\ 1 \end{bmatrix} \alpha_1^2\alpha_3 \;\middle|\; \right. \\ \left. \alpha_2 - 0.5\alpha_1\alpha_3 + 0.5\alpha_1^2 = 0.5,\ \alpha_1, \alpha_2, \alpha_3 \in [-1,1] \right\}.\end{aligned}$$

A more detailed example for constrained polynomial zonotopes is provided in Sec. 10.1.6 and in the file *examples/contSet/example_conPolyZono.m* in the CORA toolbox.

#### 2.2.1.7 Capsules

A capsule $\mathcal{C} \subset \mathbb{R}^n$ is defined as the Minkowski sum (see Sec. 2.1.1.2) of a line segment $\mathcal{L}$ and a sphere $\mathcal{S}$:

$$\mathcal{C} := \mathcal{L} \oplus \mathcal{S}, \quad \mathcal{L} = \{c + g\alpha \mid \alpha \in [-1, 1]\}, \ \mathcal{S} = \{x \mid ||x||_2 \leq r\}, \tag{10}$$

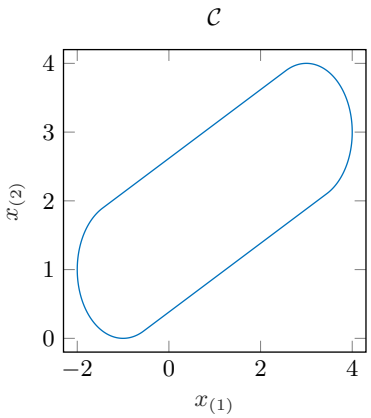
where $c, g \in \mathbb{R}^n$ represent the center and the generator of the line segment, respectively, and $r \in \mathbb{R}_{\geq 0}$ is the radius of the sphere.

Capsules are represented in CORA by the class `capsule`. An object of class `capsule` can be constructed as follows:

$$\begin{aligned}\mathcal{C} &= \texttt{capsule}(c),\\ \mathcal{C} &= \texttt{capsule}(c, g),\\ \mathcal{C} &= \texttt{capsule}(c, r),\\ \mathcal{C} &= \texttt{capsule}(c, g, r),\end{aligned}$$

where $c, g, r$ are defined as in (10). Let us demonstrate the construction of a capsule by an example:

```
% construct capsule
c = [1;2];
g = [2;1];
r = 1;

C = capsule(c,g,r);
```




A more detailed example for capsules is provided in Sec. 10.1.7 and in the file *examples/contSet/example_capsule.m* in the CORA toolbox.

#### 2.2.1.8 Zonotope Bundles

A disadvantage of zonotopes is that they are not closed under intersection, i.e., the intersection of two zonotopes does not return a zonotope in general. In order to overcome this disadvantage, zonotope bundles are introduced in [42]. Given a finite set of zonotopes $\mathcal{Z}_i \subset \mathbb{R}^n$, a zonotope bundle is defined as

$$\mathcal{ZB} := \bigcap_{i=1}^{s} \mathcal{Z}_i, \tag{11}$$

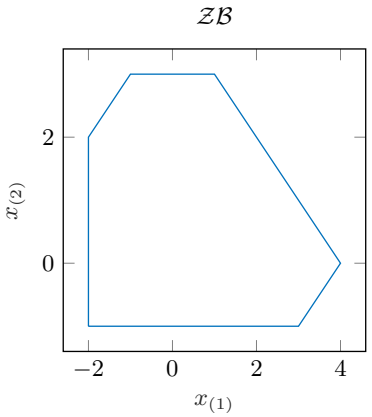
i.e., the intersection of the zonotopes $\mathcal{Z}_i$. Note that the intersection is not computed, but the zonotopes $\mathcal{Z}_i$ are stored in a list, which we write as $\mathcal{ZB} = \{\mathcal{Z}_1, \ldots, \mathcal{Z}_s\}$.

Zonotope bundles are represented in CORA by the class `zonoBundle`. An object of class `zonoBundle` can be constructed as follows:

$$\mathcal{ZB} = \texttt{zonoBundle}(\{\mathcal{Z}_1, \ldots, \mathcal{Z}_s\}),$$

where the list of zonotopes $\{\mathcal{Z}_1, \ldots, \mathcal{Z}_s\}$ is represented as a MATLAB cell array. Let us demonstrate the construction of a `zonoBundle` object by an example:

```
% construct zonotopes
zono1 = zonotope([1 3 0; 1 0 2]);
zono2 = zonotope([0 2 2; 0 2 -2]);

% construct zonotope bundle
list = {zono1,zono2};

zB = zonoBundle(list);
```




A more detailed example for zonotope bundles is provided in Sec. 10.1.8 and in the file *examples/contSet/example_zonoBundle.m* in the CORA toolbox. In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5), the class `zonoBundle` supports additional methods, which are listed in Appendix A.7.

#### 2.2.1.9 Constrained Zonotopes

An extension of zonotopes described in Sec. 2.2.1.1 are constrained zonotopes, which are introduced in [34]. A constrained zonotope is defined as a zonotope with additional equality constraints on the factors $\beta_i$:

$$\mathcal{Z}_c := \left\{ c + G\beta \;\middle|\; \|\beta\|_\infty \leq 1, A\beta = b \right\}, \tag{12}$$

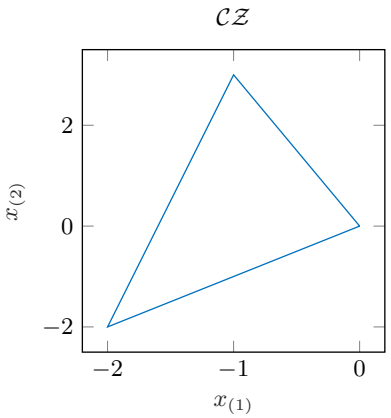
where $c \in \mathbb{R}^n$ is the zonotope center, $G \in \mathbb{R}^{n \times p}$ is the zonotope generator matrix and $\beta \in \mathbb{R}^p$ is the vector of zonotope factors. The equality constraints are parametrized by the matrix $A \in \mathbb{R}^{q \times p}$ and the vector $b \in \mathbb{R}^q$. Constrained zonotopes are able to describe arbitrary polytopes, and are therefore a more general set representation than zonotopes. The main advantage compared to a polytope representation using inequality constraints (see Sec. 2.2.1.4) is that constrained zonotopes inherit the excellent scaling properties of zonotopes for increasing state-space dimensions, since constrained zonotopes are also based on a generator representation for sets.

Constrained zonotopes are represented in CORA by the class `conZonotope`. An object of class `conZonotope` can be constructed as follows:

$$\mathcal{Z}_c = \texttt{conZonotope}(c, G, A, b),$$
$$\mathcal{Z}_c = \texttt{conZonotope}(Z, A, b),$$

where $Z = [c, G]$, and $c, G, A, b$ are defined as in (12). Let us demonstrate the construction of a constrained zonotope by an example:

```
% construct constrained zonotope
c = [0;0];
G = [1 0 1; 1 2 -1];
A = [-2 1 -1];
b = 2;

cZ = conZonotope(c,G,A,b);
```



The unconstrained zonotope from this example is visualized in Fig. 5, and the equality constraints in Fig. 6.

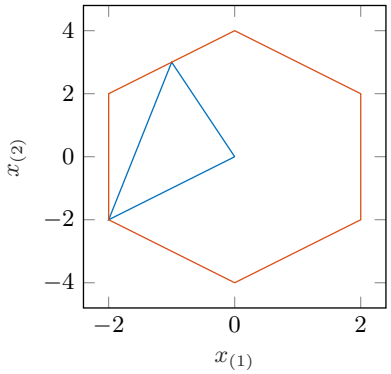



Figure 5: Zonotope (red) and the corresponding constrained zonotope (blue).

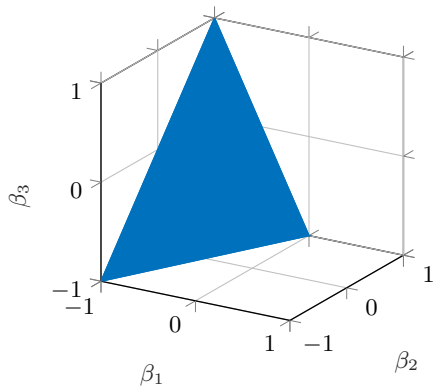



Figure 6: Visualization of the equality constraints of the constrained zonotope.

A more detailed example for constrained zonotopes is provided in Sec. 10.1.9 and in the file *examples/contSet/example_conZonotope.m* in the CORA toolbox. In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5), the class `conZonotope` supports additional methods, which are listed in Appendix A.8.

#### 2.2.1.10 Spectrahedral Shadows

Spectrahedral shadows can be seen as the semidefinite generalization of polytopes, and can represent a large variety of convex sets. In particular, every convex set representation (e.g., zonotopes, intervals, ellipsoids, polytopes, capsules, zonotope bundles, and constrained zonotopes) implemented in CORA can be represented as a spectrahedral shadow. Formally, a spectrahedral shadow $\mathcal{SPS} \subseteq \mathbb{R}^n$ is defined as

$$\mathcal{SPS} := \left\{ c + \sum_{i=1}^{p} \beta_i g^{(i)} \;\middle|\; A_{(0)} + \sum_{i=1}^{p} \beta_i A_{(i)} \succeq 0 \right\}, \tag{13}$$

where $c \in \mathbb{R}^n$ is the center vector, $g^{(i)} \in \mathbb{R}^n$ are the generators, $A_{(i)}$ are symmetric coefficient matrices, and for a matrix $M$ the expression $M \succeq 0$ means that $M$ is positive semidefinite. If the matrix $G$ with columns $g^{(i)}$ is the identity matrix, then $\mathcal{SPS}$ is also called a spectrahedron.

As shown in [43, Lemma 4.1.5.], non-empty spectrahedral shadows can equivalently be defined as

$$\mathcal{SPS} := \left\{ x \in \mathbb{R}^n \;\middle|\; \exists y \in \mathbb{R}^q, A_{(0)} + \sum_{i=1}^{p} x_i A_{(i)} + \sum_{j=1}^{q} y_j B_{(j)} \succeq 0 \right\}, \tag{14}$$

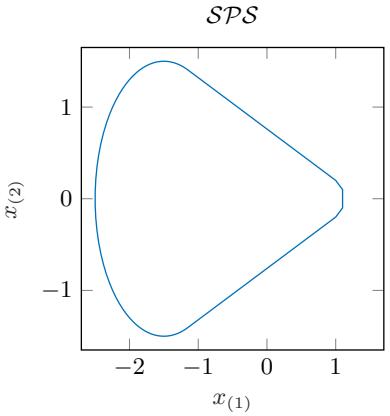
where $A_{(i)}$ and $B_{(j)}$ are symmetric coefficient matrices. We call this representation the existential sum representation.

Spectrahedral shadows are represented in CORA by the class `spectraShadow`. An object of class `spectraShadow` can be constructed as follows:

$$\begin{aligned}
\mathcal{SPS} &= \texttt{spectraShadow}(A), \\
\mathcal{SPS} &= \texttt{spectraShadow}(A, c), \\
\mathcal{SPS} &= \texttt{spectraShadow}(A, c, G), \\
\mathcal{SPS} &= \texttt{spectraShadow}(E),
\end{aligned}$$

where $G = [g^{(1)}, \dots, g^{(p)}]$, with $c, g^{(i)}$ defined as in (13) (if they are not specified, $c$ is chosen to be the origin, and $G$ the identity matrix). The matrix $A$ is the horizontal concatenation of the coefficient matrices $A_{(0)}, \dots, A_{(p)}$, i.e., $A = \begin{bmatrix} A_{(0)} & A_{(1)} & \dots & A_{(p)} \end{bmatrix}$. On the other hand, $E$ is a $1 \times 2$ cell array where the first element is a matrix $A$ that is the concatenation of the coefficient matrices matrices $A_{(0)}, \dots, A_{(p)}$, i.e., $A = \begin{bmatrix} A_{(0)} & A_{(1)} & \dots & A_{(p)} \end{bmatrix}$, and the second element is a matrix $B$ that is the concatenation of the additional coefficient matrices $B_{(0)}, \dots, B_{(q)}$, i.e., $B = \begin{bmatrix} B_{(0)} & B_{(1)} & \dots & B_{(q)} \end{bmatrix}$. Let us demonstrate the construction of a spectrahedral shadow by an example:

```
% create two simple
% spectrahedral shadows
% an ellipsoid, using the first
% type of instantiation
A0 = eye(3);
A1 = [0 1 0; 1 0 0; 0 0 0];
A2 = [0 0 1; 0 0 0; 1 0 0];
A = [A0 A1 A2];
c = [-1.5;0];
G = [1 0;0 1.5];

SpS_ellipsoid = spectraShadow(A,c,G);

% a small zonotope, using the
% second type of instantiation
A0 = diag([-19 1 11 21 1 -9]);
A1 = diag([10 10 0 -10 -10 0]);
A2 = (10/3)*diag([1 -1 2 -1 1 -2]);
A = [A0 A1 A2];

B = 0.5774*diag([-1 1 1 1 -1 -1]);

ESumRep = {A, B};

SpS_zonotope = spectraShadow(ESumRep);

% we can construct more complicated sets
% based on the two above, such as the
% convex hull of the two previous
% spectrahedral shadows
SpS = convHull(SpS_ellipsoid, SpS_zonotope);
```




A more detailed example for spectrahedral shadows is provided in Sec. 10.1.10 and in the file *examples/contSet/example_spectraShadow.m* in the CORA toolbox.

#### 2.2.1.11 Probabilistic Zonotopes

Probabilistic zonotopes have been introduced in [44] for stochastic verification. A probabilistic zonotope has the same structure as a zonotope, except that the values of some $\beta_i$ in (3) are bounded by the interval $[-1, 1]$, while others are subject to a normal distribution[16]. Given pairwise independent Gaussian-distributed random variables $\mathcal{N}(\mu, \Sigma)$ with expected value $\mu$ and covariance matrix $\Sigma$, one can define a Gaussian zonotope with certain mean:

$$\mathcal{Z}_g = c + \sum_{i=1}^{q} \mathcal{N}^{(i)}(0, 1) \cdot \underline{g}^{(i)},$$

[16] Other distributions are conceivable, but not implemented.

where $\underline{g}^{(1)}, \ldots, \underline{g}^{(q)} \in \mathbb{R}^n$ are the generators, which are underlined in order to distinguish them from generators of regular zonotopes. Gaussian zonotopes are denoted by a subscripted g: $\mathcal{Z}_g = (c, \underline{g}^{(1\ldots q)})$.

A Gaussian zonotope with uncertain mean $\mathscr{Z}$ is defined as a Gaussian zonotope $\mathcal{Z}_g$, where the center is uncertain and can have any value within a zonotope $\mathcal{Z}$, which is denoted by

$$\mathscr{Z} := \mathcal{Z} \boxplus \mathcal{Z}_g, \quad \mathcal{Z} = (c, g^{(1\ldots p)}),\ \mathcal{Z}_g = (0, \underline{g}^{(1\ldots q)}), \tag{15}$$

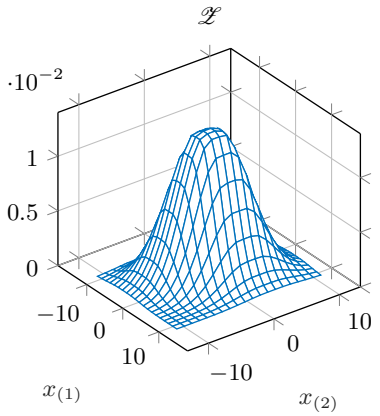
or in short by $\mathscr{Z} = (c, g^{(1\ldots p)}, \underline{g}^{(1\ldots q)})$. If the probabilistic generators can be represented by the covariance matrix $\Sigma$ ($q > n$) as shown in [44, Proposition 1], one can also write $\mathscr{Z} = (c, g^{(1\ldots p)}, \Sigma)$.

Probabilistic zonotopes are represented in CORA by the class `probZonotope`. An object of class `probZonotope` can be constructed as follows:

$$\mathscr{Z} = \texttt{probZonotope}(Z, \underline{G}),$$

where $Z = [c, g^{(1)}, \ldots, g^{(p)}]$, $\underline{G} = [\underline{g}^{(1)}, \ldots, \underline{g}^{(q)}]$, and $c, g^{(i)}, \underline{g}^{(i)}$ are defined as in (15). Let us demonstrate the construction of a probabilistic zonotope by an example:

```
% construct probabilistic zonotope
c = [0;0];
G = [1 0;0 1];
G_ = [3 2; 3 -2];

probZ = probZonotope([c,G],G_);
```



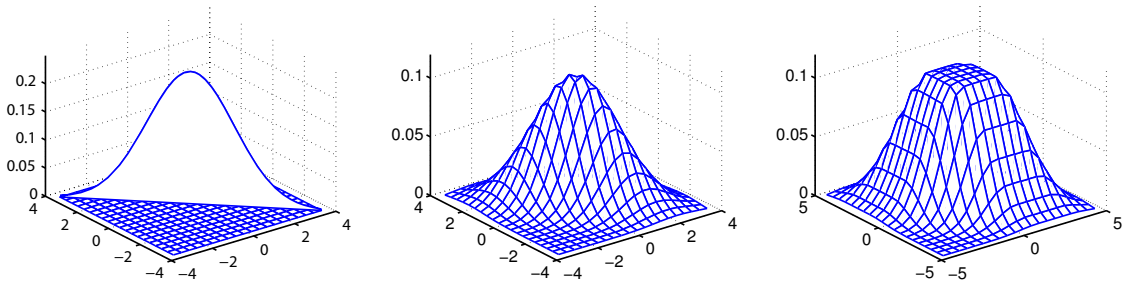

A more detailed example for probabilistic zonotopes is provided in Sec. 10.1.11 and in the file *examples/contSet/example_probZonotope.m* in the CORA toolbox.

As a probabilistic zonotope $\mathscr{Z}$ is neither a set nor a random vector, there does not exist a probability density function describing $\mathscr{Z}$. However, one can obtain an enclosing probabilistic hull which is defined as $\bar{f}_{\mathscr{Z}}(x) = \sup\left\{ f_{\mathcal{Z}_g}(x) \middle| E[\mathcal{Z}_g] \in Z \right\}$, where $E[\,]$ returns the expectation and $f_{\mathcal{Z}_g}(x)$ is the probability density function (PDF) of $\mathcal{Z}_g$. Combinations of sets with random vectors have also been investigated, e.g., in [45]. Analogously to a zonotope, it is shown in Fig. 7 how the enclosing probabilistic hull (EPH) of a Gaussian zonotope with two non-probabilistic and two probabilistic generators is built step-by-step from left to right.

In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5), the class `probZonotope` supports additional methods, which are listed in Appendix A.9.

### 2.2.2 Auxiliary Set Representations

Next, we introduce some additional set representations. These set representations are mainly used in CORA to represent guard sets for hybrid systems (see Sec. 4.3).



(a) PDF of $(0, \underline{g}^{(1)})$. (b) PDF of $(0, \underline{g}^{(1,2)})$. (c) EPH of $(0, g^{(1\ldots2)}, \underline{g}^{(1\ldots2)})$.

Figure 7: Construction of a probabilistic zonotope.

#### 2.2.2.1 Empty Set

A empty set does not contain any elements. This set representation $\mathcal{O} \subset \mathbb{R}^n$ is defined as follows:

$$\mathcal{O} = \emptyset. \tag{16}$$

Empty sets are represented in CORA by the class `emptySet`. An object of class `emptySet` can be constructed as follows:

$$\mathcal{O} = \texttt{emptySet}(n),$$

where $n$ is the number of dimensions of the set. Empty sets are used in hybrid systems to model instant transitions.

#### 2.2.2.2 Fullspace

The fullspace contains all elements. This set representation $\mathcal{FS} = \mathbb{R}^n$ is defined as follows:

$$\mathcal{FS} = \mathbb{R}^n. \tag{17}$$

Fullspaces are represented in CORA by the class `fullSpace`. An object of class `fullSpace` can be constructed as follows:

$$\mathcal{FS} = \texttt{fullSpace}(n),$$

where $n$ is the number of dimensions of the set. Fullspaces are used in hybrid systems to model unbounded invariant sets.

#### 2.2.2.3 Level Sets

A nonlinear level set $\mathcal{LS} \subset \mathbb{R}^n$ is defined as

$$\mathcal{LS} = \{x \mid f(x) = 0\}, \tag{18}$$
$$\mathcal{LS} = \{x \mid f(x) < 0\}, \text{ or} \tag{19}$$
$$\mathcal{LS} = \{x \mid f(x) \leq 0\}, \tag{20}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a Lipschitz continuous function. Level sets are represented in CORA by the class `levelSet`. An object of class `levelSet` can be constructed as follows:

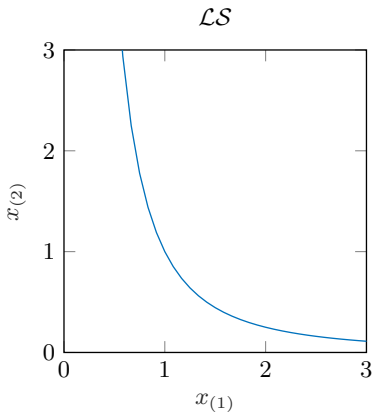
$$\mathcal{LS} = \texttt{levelSet}(f(\cdot), \texttt{vars}, \texttt{op}),$$

where

- $f : \mathbb{R}^n \to \mathbb{R}$ is the nonlinear function that defines the level set (see (18),(19), and (20)). The function is specified as a symbolic MATLAB function.
- `vars` is a vector containing the symbolic variables of the function $f(\cdot)$.
- `op` $\in$ {'==','<','<='} defines the type of level set ((18),(19), or (20), respectively).

Let us demonstrate the construction of a level set by an example:

```
% construct level set
vars = sym('x',[2,1]);
f = 1/vars(1)^2 - vars(2);
op = '==';

ls = levelSet(f,vars,op);
```




A more detailed example for level sets is provided in Sec. 10.1.12 and in the file *examples/contSet/example_levelSet.m* in the CORA toolbox. In addition to the standard set operations described in Sec. 2.1, the class `levelSet` supports additional methods, which are listed in Appendix A.10.

### 2.2.3 Set Representations for Range Bounding

For general nonlinear functions it is often infeasible or impossible to exactly determine its minimum and maximum on a certain domain. Therefore, one often tightly encloses the minimum and maximum by range bounding. Given a nonlinear function $f : \mathbb{R}^n \to \mathbb{R}$ and a domain $\mathcal{D} \subset \mathbb{R}^n$, the range bounding operation `B` returns a tight enclosure of the function values:

$$\texttt{B}(f(x), \mathcal{D}) \supseteq \left[\min_{x \in \mathcal{D}} f(x),\ \max_{x \in \mathcal{D}} f(x)\right]. \tag{21}$$

There exist many different ways to implement the range bounding operation `B` in (21). The simplest method is to apply *interval arithmetic* [46], for which the `interval` class (see Sec. 2.2.1.2) can be used. A detailed description how *interval arithmetic* is implemented in CORA is provided in [11]. However, while interval arithmetic is fast, it often results in quite conservative bounds. We therefore additionally implemented Taylor models [47] by the class `taylm` (see Sec. 2.2.3.1), affine arithmetic [48] by the class `affine` (see Sec. 2.2.3.2), and a combination of several methods by the class `zoo` (see Sec. 2.2.3.3).

Let us first demonstrate range bounding for the nonlinear function $f(x) = \sin(x_1)x_2 + x_1^2$ within the domain $x_1 \in [-1, 2]$, $x_2 \in [0, 1]$. Bounds using interval arithmetic can be computed as follows:

```
% function f(x)
f = @(x) sin(x(1))*x(2) + x(1)^2;

% domain D for x
D = interval([-1;0],[2;1]);

% compute bounds
res = f(D)
```

```
Command Window:

res =

 [-0.84147,5.00000]
```

#### 2.2.3.1 Taylor Models

Taylor models [47, 49–51] can be used to obtain rigorous bounds of functions that are often tighter than the ones obtained by interval arithmetic. A Taylor model $\mathcal{T}(x)$ is defined as

$$\mathcal{T}(x) = \{p(x) + y \mid y \in \mathcal{I}\}, \tag{22}$$

where $p : \mathbb{R}^p \to \mathbb{R}^n$ is a polynomial function and $\mathcal{I} \subset \mathbb{R}^n$ is an interval (see Sec. 2.2.1.2). For range bounding, the possible values for the variable $x$ are usually restricted by an interval domain $\mathcal{D} \subset \mathbb{R}^p$ (see (21)).

To enclose a nonlinear function with a Taylor model, a Taylor series expansion of the function is computed:

$$f(x) \approx f(x^*) + \left.\frac{\partial f}{\partial x}\right|_{x^*} (x - x^*) + \left.\frac{\partial^2 f}{\partial x^2}\right|_{x^*} (x - x^*)^2 + \dots \ .$$

Let us consider the nonlinear function $f(x) = \cos(x)$ as an example. By computing a second-order Taylor series expansion at the expansion point $x^* = 0$, the function $f(x)$ on the domain $x \in [-1, 1]$ can be enclosed by the Taylor model

$$\mathcal{T}(x) := \left\{1 - 0.5x^2 + y \;\middle|\; y \in [-0.15, 0.15]\right\}, \tag{23}$$

which is visualized in Fig. 8.

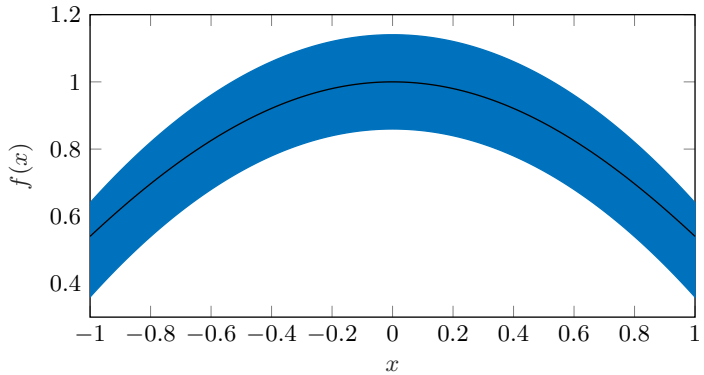



Figure 8: Function $f(x) = \cos(x)$ (black) and the enclosing Taylor model $\mathcal{T}(x)$ in (23) (blue).

Taylor models are represented by the class `taylm`. An object of class `taylm` can be constructed as follows:

$$\mathcal{T}(x) = \texttt{taylm}(\mathcal{D}),$$
$$\mathcal{T}(x) = \texttt{taylm}(\mathcal{D}, \texttt{maxOrder}, \texttt{name}, \texttt{optMethod}, \texttt{tolerance}, \texttt{eps}),$$

where $\mathcal{D} \subset \mathbb{R}^p$ is the interval domain for the variable $x$. The domain $\mathcal{D}$ is defined by an object of class `interval` (see Sec. 2.2.1.2). The additional optional parameters are defined as follows:

- `maxOrder`: Maximum polynomial degree of the monomials in the polynomial part of the Taylor model. Monomials with a degree larger than `maxOrder` are enclosed and added to the interval remainder. Further, $q =$ `maxOrder` is used for the implementation of the formulas listed in [12, Appendix A].
- `name:` String or cell array of strings defining the names for the variables. Unique names are important since Taylor models explicitly consider dependencies between the variables.
- `optMethod`: Method used to calculate the bounds of the Taylor model objects. The available methods are 'int' (interval arithmetic, default), 'bnb' (branch and bound algorithm, see [12, Sec. 2.3.2]), 'bnbAdv' (branch and bound with Taylor model re-expansion) and 'linQuad' (optimization with Linear Dominated Bounder and Quadratic Fast Bounder, see [12, Sec. 2.3.3])
- `tolerance:` Minimum absolute value of the monomial coefficients in the polynomial part of the Taylor model. Monomials with a coefficient whose absolute value is smaller than `tolerance` are enclosed and added to the interval remainder.
- `eps:` Termination tolerance $\epsilon$ for the branch and bound algorithm from [12, Sec. 2.3.2] and for the algorithm based on the Linear Dominated Bounder and the Quadratic Fast Bounder from [12, Sec. 2.3.3].

CORA also supports to create Taylor models from symbolic functions. A detailed description of this is provided in Appendix A.11.1.

Let us demonstrate Taylor models by an example:

```
% function f(x)
f = @(x) sin(x(1))*x(2) + x(1)^2;

% create Taylor model
D = interval([-1;0],[2;1]);

tay = taylm(D,10,'x','linQuad');

% compute bounds
res = interval(f(tay))
```

```
Command Window:

res =

 [-0.23256,4.90940]
```

A more detailed example for Taylor models is provided in Sec. 10.1.13 and in the file *examples/contSet/example_taylm.m* in the CORA toolbox. A detailed description of how Taylor models are treated in CORA can be found in [12]. Furthermore, a list of operations that are implemented for the class `taylm` is provided in Appendix A.11.

#### 2.2.3.2 Affine

Affine arithmetic uses affine forms, i.e., first-order polynomials consisting of a vector $x \in \mathbb{R}^n$ and noise symbols $\epsilon_i \in [-1, 1]$ (see e.g., [48]):

$$\hat{x} = x_0 + \epsilon_1 x_1 + \epsilon_2 x_2 + \ldots + \epsilon_p x_p.$$

The possible values of $\hat{x}$ lie within a zonotope [52].

Affine arithmetic is implemented by the class `affine`. Since we only consider intervals as inputs and outputs, we realized affine arithmetic as Taylor models of first order. The class `affine` therefore inherits all methods from the class `taylm` and does not implement any functionality on its own. The main purpose of the class `affine` is to provide a convenient and easy-to-use interface for the user. An object of class `affine` can be constructed as follows:

$$\mathcal{A}(x) = \texttt{affine}(\mathcal{D}),$$
$$\mathcal{A}(x) = \texttt{affine}(\mathcal{D}, \texttt{order}, \texttt{name}, \texttt{optMethod}, \texttt{tolerance}, \texttt{eps}),$$

where the input arguments are identical to the ones for the class `taylm` (see Sec. 2.2.3.1). Let us demonstrate the class `affine` by an example:

```
% function f(x)
f = @(x) sin(x(1)).*x(2) + x(1)^2;

% create affine object
D = interval([-1;0],[2;1]);

aff = affine(D);

% compute bounds
res = interval(f(aff))
```

```
Command Window:

res =

 [-3.69137,6.74245]
```

A more detailed example for the class `affine` is provided in Sec. 10.1.14 and in the file *examples/contSet/example_affine.m* in the CORA toolbox.

#### 2.2.3.3 Zoo

When it comes to range bounding, it is often better to use several simple range bounding methods in parallel and intersect the result, instead of tuning one method towards high accuracy. This is

demonstrated by the numerical examples shown in [12] and by the code example in Sec. 10.1.15. To facilitate mixing different range bounding techniques, we created the class `zoo` in which one can specify the methods to be combined. An object of class `zoo` can be constructed as follows:

$$\mathcal{Z}(x) = \texttt{zoo}(\mathcal{D}, \texttt{methods}),$$
$$\mathcal{Z}(x) = \texttt{zoo}(\mathcal{D}, \texttt{methods}, \texttt{name}, \texttt{maxOrder}, \texttt{tolerance}, \texttt{eps}),$$

where all input arguments except of `methods` are identical to the ones for the class `taylm` (see Sec. 2.2.3.1). The argument `methods` is a cell array containing strings that describe the range bounding methods that are combined. The following range bounding methods are available:

- `'interval'` – Interval arithmetic (see Sec. 2.2.1.2).
- `'affine(int)'` – Affine arithmetic; the bounds of the affine objects are calculated with interval arithmetic (see Sec. 2.2.3.2).
- `'affine(bnb)'` – Affine arithmetic; the bounds of the affine objects are calculated with the branch and bound algorithm (see Sec. 2.2.3.2).
- `'affine(bnbAdv)'` – Affine arithmetic; the bounds of the affine objects are calculated with the advanced branch and bound algorithm (see Sec. 2.2.3.2).
- `'affine(linQuad)'` – Affine arithmetic; the bounds of the affine objects are calculated with the algorithm that is based on the Linear Dominated Bounder and the Quadratic Fast Bounder (see Sec. 2.2.3.2).
- `'taylm(int)'` – Taylor models; the bounds of the Taylor models are calculated with interval arithmetic (see Sec. 2.2.3.1).
- `'taylm(bnb)'` – Taylor models; the bounds of the Taylor models are calculated with the branch and bound algorithm (see Sec. 2.2.3.1).
- `'taylm(bnbAdv)'` – Taylor models; the bounds of the Taylor models are calculated with the advanced branch and bound algorithm (see Sec. 2.2.3.1).
- `'taylm(linQuad)'` – Taylor models; the bounds of the Taylor models are calculated with the algorithm that is based on the Linear Dominated Bounder and the Quadratic Fast Bounder (see Sec. 2.2.3.1).

All functions that are implemented for class `taylm` are also available for the class `zoo`. Let us demonstrate the class `zoo` by an example:

```
% function f(x)
f = @(x) sin(x(1)).*x(2) + x(1)^2;

% create zoo object
D = interval([-1;0],[2;1]);
methods = {'interval','taylm(linQuad)'};

Z = zoo(D,methods);

% compute bounds
res = interval(f(Z))
```

```
Command Window:

res =

 [-0.23983,4.92298]
```

A more detailed example for the class `zoo` is provided in Sec. 10.1.15 and in the file *examples/contSet/example_zoo.m* in the CORA toolbox.

# 3 Matrix Set Representations and Operations

Besides vector sets as introduced in the previous section, it is often useful to represent sets of possible matrices. This occurs for instance when a linear system has uncertain parameters as described later in Sec. 4.2.2. CORA supports the following matrix set representations:

- Matrix polytope (Sec. 3.2.1).
- Matrix zonotope (Sec. 3.2.2, specialization of a matrix polytope).
- Interval matrix (Sec. 3.2.3, specialization of a matrix zonotope).

Note that we use the term *matrix polytope* instead of *polytope matrix*. The reason is that the analogous term *vector polytope* makes sense, while *polytope vector* can be misinterpreted as a vertex of a polytope. We do not use the term *matrix interval* since the term *interval matrix* is already established.

For each matrix set representation, the conversion to all other matrix set computations is implemented. Of course, conversions to specializations are realized in an over-approximative way, while the other direction is computed exactly (see Tab. 6). In order to convert a matrix set, it is sufficient to pass the current matrix set object to the class constructor of the target matrix set representation, as demonstrated by the following example:

```
% interval matrix
C = [0 1;0 -2.5];
D = [0 0;0 0.5];
intMat = intervalMatrix(C,D);

% conversion to other matrix set representations
matZono = matZonotope(intMat);
matPoly = matPolytope(intMat);
```

Table 6: Matrix set conversions supported by CORA. The row headers represent the original matrix set representation and the column headers the target matrix set representation after conversion. The shortcuts e (exact conversion) and o (over-approximation) are used.

| | **matPolytope** | **matZonotope** | **intervalMatrix** |
|---|---|---|---|
| **matPolytope** (Sec. 3.2.1) | - | o | o |
| **matZonotope** (Sec. 3.2.2) | e | - | o |
| **intervalMatrix** (Sec. 3.2.3) | e | e | - |

We first introduce importrant operations for matrix sets in Sec. 3.1 before we describe the matrix set representations implemented in detail in Sec. 3.2.

## 3.1 Matrix Set Operations

This section describes the implemented standard operations for matrix sets.

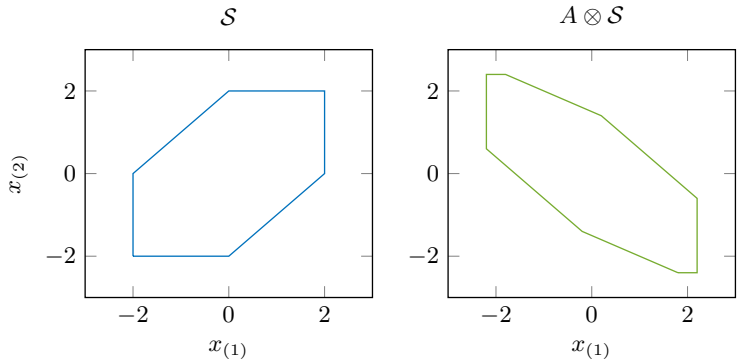
### 3.1.1 mtimes

The method `mtimes`, which overloads the $*$ operator, implements the multiplication of two matrix sets or the multiplication of a matrix set with a vector set, depending on the input arguments. Given two matrix sets $\mathcal{A}_1, \mathcal{A}_2 \subset \mathbb{R}^{n \times n}$ and a vector set $\mathcal{S} \subset \mathbb{R}^n$, the method `mtimes` computes

$$\texttt{mtimes}(\mathcal{A}_1, \mathcal{A}_2) = \mathcal{A}_1 \otimes \mathcal{A}_2 = \{A_1 \cdot A_2 \mid A_1 \in \mathcal{A}_1,\ A_2 \in \mathcal{A}_2\},$$
$$\texttt{mtimes}(\mathcal{A}_1, \mathcal{S}) = \mathcal{A}_1 \otimes \mathcal{S} = \{A_1 \cdot s \mid A_1 \in \mathcal{A}_1,\ s \in \mathcal{S}\}.$$

If the corresponding matrix set representation is not closed under multiplication, `mtimes` returns an over-approximation. Let us demonstrate the method `mtimes` by an example:

```
% vector set
S = zonotope([0 1 1 0; ...
              0 1 0 1]);

% matrix set
C = [1 0; -1 0.5];
D = [0.1 0; 0 0.2];
A = intervalMatrix(C,D);

% linear transformation
res = A * S;
```




### 3.1.2 plus

The method `plus`, which overloads the $+$ operator, implements the Minkowski sum of two matrix sets. Given two matrix sets $\mathcal{A}_1, \mathcal{A}_2 \subset \mathbb{R}^{n \times n}$, their Minkowski sum is defined as

$$\texttt{plus}(\mathcal{A}_1, \mathcal{A}_2) = \mathcal{A}_1 \oplus \mathcal{A}_2 = \{A_1 + A_2 \mid A_1 \in \mathcal{A}_1,\ A_2 \in \mathcal{A}_2\}.$$

If the corresponding matrix set representation is not closed under Minkowski sum, `plus` returns an over-approximation. Let us demonstrate the method `plus` by an example:

```
% matrix sets
A1 = intervalMatrix([0 1;2 3],[1 2;0 1]);
A2 = intervalMatrix([3 2;2 2],[0 1;1 0]);

% Minkowski addition
res = A1 + A2
```

```
Command Window:

res =
[2.000,4.000] [0.000,6.000]
[3.000,5.000] [4.000,6.000]
```

### 3.1.3 expm

Given a matrix set $\mathcal{A} \subset \mathbb{R}^{n \times n}$, the method `expm` computes a tight enclosure of the matrix exponential

$$\texttt{expm}(\mathcal{A}) \supseteq e^{\mathcal{A}} = \sum_{i=0}^{\infty} \frac{\mathcal{A}^k}{k!} .$$

The number of Taylor terms $\eta$ used for the calculation of the matrix exponential (see [33, Theorem 3.2]) can be specified as an additional input argument:

$$\texttt{expm}(\mathcal{A}, \eta) \supseteq e^{\mathcal{A}}.$$

The computation of a tight enclosure of the matrix exponential for matrix sets is essential for reachability analysis of linear parametric systems (see Sec. 4.2.2). Let us demonstrate the method `expm` by an example:

```
% matrix set
C = [0 1;0 -2.5];
D = [0 0;0 0.5];
A = intervalMatrix(C,D);

% matrix exponential
eA = expm(A)
```

```
Command Window:

res =
[1.00000,1.00000] [-1.21072,1.95859]
[0.00000,0.00000] [-5.25685,5.44556]
```

### 3.1.4 vertices

Given a matrix set $\mathcal{A} \subset \mathbb{R}^{n \times n}$, the method `vertices` computes its vertices $V_1, \ldots, V_q$, $V_i \in \mathbb{R}^{n \times n}$:

$$\texttt{vertList} = \texttt{vertices}(\mathcal{A}),$$

where `vertList` is a MATLAB cell array that stores the vertices $V_i$. Let us demonstrate the method `vertices` by an example:

```
% matrix set
C = [0 1;3 2];
G(:,:,1) = [1 2;0 1];

A = matZonotope(C,G);

% compute vertices
res = vertices(A)
```

```
Command Window:

res{1} =
 -1.0000   -1.0000
  3.0000    1.0000

res{2} =
  1.0000    3.0000
  3.0000    3.0000
```

## 3.2 Matrix Set Representations

This section describes the different matrix set representations implemented in CORA.

### 3.2.1 Matrix Polytopes

A matrix polytope is defined analogously to a V-polytope (see Sec. 2.2.1.4):

$$\mathcal{A}_{[p]} = \left\{ \sum_{i=1}^{r} \alpha_i V^{(i)} \;\middle|\; \alpha_i \in \mathbb{R}, \alpha_i \geq 0, \sum_i \alpha_i = 1 \right\}, \quad V^{(i)} \in \mathbb{R}^{n \times m}. \tag{24}$$

The matrices $V^{(i)}$ are also called vertices of the matrix polytope. When substituting the matrix vertices by vector vertices $v^{(i)} \in \mathbb{R}^n$, one obtains a V-polytope (see Sec. 2.2.1.4).

Matrix polytopes are implemented in CORA by the class `matPolytope`. An object of class `matPolytope` can be constructed as follows:

$$\mathcal{A}_{[p]} = \texttt{matPolytope}(\texttt{V}),$$

where `V` is a three-dimensional array $(n \times m \times r)$ that stores the vertices $V^{(i)}$, see (24), of the matrix polytope.

Let us demonstrate the construction of a `matPolytope` object by an example:

```
% vertices
V(:,:,1) = [1 2; 0 1];
V(:,:,2) = [1 3; -1 2];

% matrix polytope
mp = matPolytope(V);
```

A more detailed example for matrix polytopes is provided in Sec. 10.2.1 and in the file *examples/matrixSet/example_matPolytope.m* in the CORA toolbox. Furthermore, a list of methods for the class `matPolytope` is provided in Appendix B.1.

### 3.2.2 Matrix Zonotopes

A matrix zonotope is defined analogously to zonotopes (see Sec. 2.2.1.1):

$$\mathcal{A}_{[z]} = \left\{ C + \sum_{i=1}^{\kappa} p_i G^{(i)} \middle| p_i \in [-1, 1] \right\}, \quad G^{(i)} \in \mathbb{R}^{n \times m}, \tag{25}$$

and is written in short form as $\mathcal{A}_{[z]} = (C, G^{(1)}, \ldots, G^{(\kappa)})$, where the first matrix is referred to as the *matrix center* and the other matrices as *matrix generators*. The order of a matrix zonotope is defined as $\rho = \kappa / n$. When exchanging the matrix generators by vector generators $g^{(i)} \in \mathbb{R}^{n \cdot m}$, one obtains a zonotope (see e.g., [32]).

Matrix zonotopes are implemented by the class `matZonotope`. An object of class `matZonotope` can be constructed as follows:

$$\mathcal{A}_{[z]} = \texttt{matZonotope}(C, \texttt{G}),$$

where `G` is a three-dimensional array $(n \times m \times \kappa)$ that stores the generator matrices, see (25).

Let us demonstrate the construction of a `matZonotope` object by an example:

```
% matrix center
C = [0 0; 0 0];

% matrix generators
G(:,:,1) = [1 3; -1 2];
G(:,:,2) = [2 0; 1 -1];

% matrix zonotope
mz = matZonotope(C,G);
```

A more detailed example for matrix zonotopes is provided in Sec. 10.2.2 and in the file *examples/matrixSet/example_matZonotope.m* in the CORA toolbox. Furthermore, a list of methods for the class `matZonotope` is provided in Appendix B.2.

### 3.2.3 Interval Matrices

An interval matrix is a special case of a matrix zonotope and specifies the interval of possible values for each matrix element:

$$\mathcal{A}_{[i]} = [\underline{A}, \overline{A}], \quad \forall i,j : \underline{a}_{ij} \leq \overline{a}_{ij}, \quad \underline{A}, \overline{A} \in \mathbb{R}^{n \times n}.$$

The matrix $\underline{A}$ is referred to as the *lower bound* and $\overline{A}$ as the *upper bound* of $\mathcal{A}_{[i]}$.

In CORA, interval matrices are implemented by the class `intervalMatrix`. An object of class `intervalMatrix` can be constructed as follows:

$$\mathcal{A}_{[i]} = \texttt{intervalMatrix}(C, D),$$

where $C = 0.5(\overline{A} + \underline{A})$ is the center matrix and $D = 0.5(\overline{A} - \underline{A})$ is the width matrix.

Let us demonstrate the construction of an `intervalMatrix` object by an example:

```
% center matrix
C = [0 2; 3 1];

% width matrix
D = [1 2; 1 1];

% interval matrix
mi = intervalMatrix(C,D);
```

A more detailed example for interval matrices is provided in Sec. 10.2.3 and in the file *examples/matrixSet/example_intervalMatrix.m* in the CORA toolbox. Furthermore, a list of methods for the class `intervalMatrix` is provided in Appendix B.3.

# 4 Dynamic Systems and Operations

This section introduces the dynamic systems and operations on them. As in Sec. 2 on set representations, we start with the operations.

## 4.1 Dynamic System Operations

To improve the usability of CORA, all dynamic systems share a set of identical operations, such as `reach` to compute the reachable set. This subsection presents the most common, currently implemented operations.

### 4.1.1 reach

The operation `reach` computes the reachable set of a dynamic system. Let us denote the solution of a dynamic system by $\chi(t; x_0, u(\cdot), p)$, where $t \in \mathbb{R}$ is the time, $x_0 = x(t_0) \in \mathbb{R}^n$ is the initial state, $u(\cdot) \in \mathbb{R}^m$ is the system input, and $p \in \mathbb{R}^p$ is a parameter vector. The reachable set at time $t = t_f$ can be defined for a set of initial states $\mathcal{X}_0 \subset \mathbb{R}^n$, a set of input values $\mathcal{U}(t) \subset \mathbb{R}^m$, and a set of parameter values $\mathcal{P} \subset \mathbb{R}^p$, as

$$\mathcal{R}^e(t_f) = \left\{ \chi(t_f; x_0, u(\cdot), p, w) \in \mathbb{R}^n \;\middle|\; x_0 \in \mathcal{X}_0, \forall t \in [t_0, t_f] : u(t) \in \mathcal{U}(t), p \in \mathcal{P}, w \in \mathcal{W} \right\}. \quad (26)$$

Since the exact reachable set $\mathcal{R}^e(t)$ as defined in (26) cannot be computed in general, the operation `reach` computes a tight enclosure $\mathcal{R}(t) \supseteq \mathcal{R}^e(t)$.

The syntax for the operation `reach` is:

```
R = reach(sys, params, options)
,[R, res] = reach(sys, params, options, spec),
```

with input arguments

- `sys` dynamic system defined by any of the classes in Sec. 4.2 and 4.3, e.g., `linearSys`, `hybridAutomaton`, etc.
- `params` struct containing the parameter that define the reachability problem
  - `.tStart` initial time $t_0$ (default value 0)
  - `.tFinal` final time $t_f$
  - `.R0` initial set $\mathcal{X}_0$ specified by one of the set representations in Sec. 2.2.1
  - `.U` input set $\mathcal{U}$ specified as an object of class `zonotope` (see Sec. 2.2.1.1)
  - `.u` time-dependent center $u_c(t)$ of the time-varying input set $\mathcal{U}(t) := u_c(t) \oplus \mathcal{U}$ specified as a matrix for which the number of columns is identical to the number of reachability steps (optional)
  - `.paramInt` set of parameter values $\mathcal{P}$ specified as an object of class `interval` (see Sec. 2.2.1.2) (class `nonlinParamSys` only)
  - `.W` disturbance set $\mathcal{W}$ specified as an object of class `interval` (see Sec. 2.2.1.2) or `zonotope` (see Sec. 2.2.1.1 (classes `linearSys` and `linearSysDT` only)
  - `.V` set of sensor noises $\mathcal{V}$ specified as an object of class `interval` (see Sec. 2.2.1.2)or `zonotope` (see Sec. 2.2.1.1 (classes `linearSys` and `linearSysDT` only)

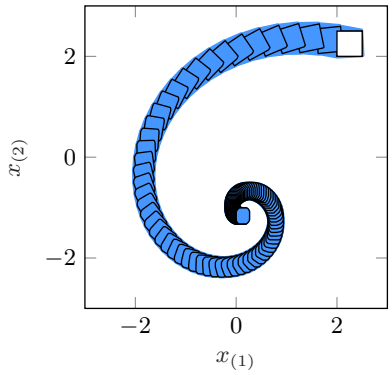
  - `.y0guess` guess for a consistent initial algebraic state (class `nonlinDASys` only, see Sec. 4.2.10.1).
  - `.startLoc` index of the initial location (class `hybridAutomaton` and `parallelHybridAutomaton` only)
  - `.finalLoc` index of the final location. Reachability analysis stops as soon as the final location is reached (class `hybridAutomaton` and `parallelHybridAutomaton` only, optional)
- `options` struct containing algorithm settings for reachability analysis. Since the settings are different for each type of dynamic system, they are documented in Sec. 4.2 and Sec. 4.3.
- `spec` object of class `specification` (see Sec. 7.3) which represents the specifications the system has to verify. Reachability analysis stops as soon as a specification is violated.

and output arguments

- `R` object of class `reachSet` (see Sec. 7.1) that stores the reachable set $\mathcal{R}(t_i)$ at time point $t_i$ and the reachable set $\mathcal{R}(\tau_i)$ for time intervals $\tau_i = [t_i, t_{i+1}]$.
- `res` Boolean flag that indicates whether the specifications are satisfied (`res` $= 1$) or not (`res` $= 0$).

Let us demonstrate the operation `reach` by an example:

```
% system dynamics
sys = linearSys([-0.7 -2;2 -0.7],[1;1],[-2;-1]);

% parameter
params.tFinal = 5;
params.R0 = zonotope(interval([2;2],[2.5;2.5]));
params.U = zonotope(interval(-0.1,0.1));

% reachability settings
options.timeStep = 0.05;
options.zonotopeOrder = 10;
options.taylorTerms = 5;

% reachability analysis
R = reach(sys,params,options);
```




### 4.1.2 reachInner

The operation `reach`, which was introduced in Sec. 4.1.1, computes a tight outer-approximation $\mathcal{R}(t) \supseteq \mathcal{R}^e(t)$ of the exact reachable set $\mathcal{R}^e(t)$ as defined in (26). For most cases computing an outer-approximation of the reachable set is sufficient. However, sometimes it also required to compute an inner-approximation $\mathcal{R}^i(t) \subseteq \mathcal{R}^e(t)$ of the exact reachable set: Inner-approximations can be used to prove that a system provably violates a given specification, they are required for conformance testing using *reachset conformance* [53], and they are very useful for controller synthesis where one often has to prove that the controlled system is guaranteed to reach a certain goal set.

The operation `reachInner` computes a tight inner-approximation $\mathcal{R}^i(t) \subseteq \mathcal{R}^e(t)$ of the exact reachable set $\mathcal{R}^e(t)$ as defined in (26). Currently, CORA only supports the computation of inner-approximations for the reachable set at certain time-points $t_i$, but not for time intervals $\tau_i = [t_i, t_{i+1}]$. Since some approaches compute an inner-approximation based on the outer-approximation of the reachable set, the operation `reachInner` returns in some cases both an

inner-approximation as well as an outer-approximation of the reachable set.

The syntax for the operation `reachInner` is:

$$[\texttt{Rin}, \texttt{Rout}] = \texttt{reachInner}(\texttt{sys}, \texttt{params}, \texttt{options}),$$
$$\texttt{Rin} = \texttt{reachInner}(\texttt{sys}, \texttt{params}, \texttt{options}),$$

with input arguments

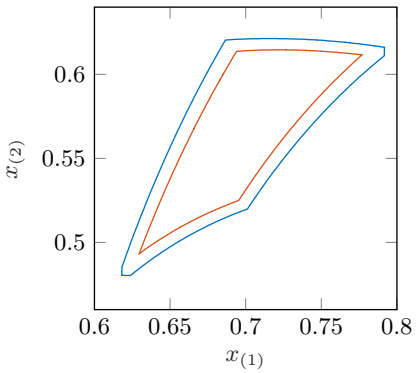
- `sys` dynamic system defined by any of the classes in Sec. 4.2, e.g., `linearSys`, `nonlinearSys`, etc.
- `params` struct containing the parameter that define the reachability problem. The parameters are identical to those for the operation `reach` (see Sec. 4.1.1).
- `options` struct containing algorithm settings for reachability analysis. Since the settings are different for each type of dynamic system, they are documented in Sec. 4.2.

and output arguments

- `Rin` object of class `reachSet` (see Sec. 7.1) that stores the inner-approximations $\mathcal{R}^i(t_i)$ of the reachable set at time points $t_i$.
- `Rout` object of class `reachSet` (see Sec. 7.1) that stores the outer-approximations $\mathcal{R}(t_i)$ of the reachable set at time points $t_i$ (class `nonlinearSys` only).

Let us demonstrate the operation `reachInner` by an example:

```
% system dynamics
f = @(x,u) [1-2*x(1) + 3/2 * x(1)^2*x(2); ...
            x(1)-3/2*x(1)^2*x(2)];

sys = nonlinearSys(f);

% parameter
params.tFinal = 1;
params.R0 = interval([0.75;0],[1;0.25]);

% reachability settings
options.algInner = 'scale';
options.timeStep = 0.001;
options.taylorTerms = 10;
options.zonotopeOrder = 50;
options.intermediateOrder = 20;
options.errorOrder = 10;

% reachability analysis
[Rin,Rout] = reachInner(sys,params,options);
```




### 4.1.3 reachBackward

The operation `reachBackward` computes the backward reachable set of a dynamic system. Let us denote the solution of a dynamic system by $\chi(t; x_0, u(\cdot), w(\cdot))$, where $t \in \mathbb{R}$ is the time, $x_0 = x(t_0) \in \mathbb{R}^n$ is the initial state, $u(\cdot) \in \mathbb{R}^m$ is the control input, and $w \in \mathbb{R}^r$ is the disturbance. Given a target set $\mathcal{X}_0 \subset \mathbb{R}^n$, a set of input values $\mathcal{U}(t) \subset \mathbb{R}^m$, a set of disturbance values $\mathcal{W}(t) \subset \mathbb{R}^r$, a time interval $\tau$, we can define two backward reachable sets, depending on the

quantifiers used for control inputs and disturbances:

$$\mathcal{R}_{AE}(-\tau) = \left\{ x_0 \in \mathbb{R}^n \mid \forall u(t) \in \mathcal{U}(t), \exists w \in \mathcal{W}, \exists t \in \tau \colon \chi(t; x_0, u(\cdot), w(\cdot)) \in \mathcal{X}_0 \right\}. \tag{27}$$

$$\mathcal{R}_{EA}(-\tau) = \left\{ x_0 \in \mathbb{R}^n \mid \exists w \in \mathcal{W}, \forall u(t) \in \mathcal{U}(t), \exists t \in \tau \colon \chi(t; x_0, u(\cdot), w(\cdot)) \in \mathcal{X}_0 \right\}. \tag{28}$$

Please note that the time interval $\tau$ may also be a single point in time. Since the exact reachable set $\mathcal{R}^e(t)$ as defined in (26) cannot be computed in general, the operation `reach` computes a tight enclosure $\mathcal{R}(t) \supseteq \mathcal{R}^e(t)$.

The syntax for the operation `reachBackward` is:

$$\texttt{R} = \texttt{reachBackward}(\texttt{sys}, \texttt{params}, \texttt{options}),$$

with input arguments

- `sys` dynamic system as an object of class `linearSys`
- `params` struct containing the parameters that define the reachability problem
  - `.tStart` initial time $t_0$ (default value 0)
  - `.tFinal` final time $t_f$
  - `.R0` initial set $\mathcal{X}_0$ specified by a polytope (see **??**)
  - `.U` input set $\mathcal{U}$ specified as an object of class `interval` (see Sec. 2.2.1.2) or `zonotope` (see Sec. 2.2.1.1)
  - `.W` disturbance set $\mathcal{W}$ specified as an object of class `interval` (see Sec. 2.2.1.2) or `zonotope` (see Sec. 2.2.1.1)
- `options` struct containing algorithm settings for reachability analysis. Since the settings are different for each type of dynamic system, they are documented in Sec. 4.2 and Sec. 4.3.
  - `.timeStep` time step size
  - `.linAlg` string specifying the used reachability algorithm following the pattern `<inner|outer>:<AE|EA>:<timepoint|timeinterval>`, where the first part refers to computing inner or outer approximations, the second part to the AE backward reachable set (27) or EA backward reachable set (28), and the third part to computing time-point or time-interval solutions.

and output arguments

- `R` object of class `reachSet` (see Sec. 7.1) that stores the backward reachable set $\mathcal{R}(t_i)$ at time point $t_i$ and the reachable set $\mathcal{R}(\tau_i)$ for time intervals $\tau_i = [t_i, t_{i+1}]$.

Let us demonstrate the operation `reachBackward` by an example:

```
% system dynamics
sys = linearSys([0 1; 0 0],[0;1],[],[],[],[],[1 0; 0 1]);

% parameter
params.tStart = 0;
params.tFinal = 3;
params.R0 = polytope(interval([-0.5;-0.5],[0.5;0.5]));
params.U = zonotope(0,1);
params.W = zonotope(zeros(2,1),0.1*eye(2));

% reachability settings
options.timeStep = 0.1;
options.linAlg = 'inner:EA:timeinterval';

% reachability analysis
R = reachBackward(sys,params,options);
```

### 4.1.4 simulate

The operation `simulate` simulates a dynamical system and returns a trajectory starting from the initial state $x_0 = x(t_0) \in \mathbb{R}^n$ for an input signal $u(t) \in \mathbb{R}^m$ and a parameter value $p \in \mathbb{R}^p$. The syntax is as follows:

$$[\texttt{t},\texttt{x}] = \texttt{simulate}(\texttt{sys},\texttt{params}),$$
$$[\texttt{t},\texttt{x},\texttt{ind}] = \texttt{simulate}(\texttt{sys},\texttt{params},\texttt{options}),$$
$$[\texttt{t},\texttt{x},\texttt{ind},\texttt{y}] = \texttt{simulate}(\texttt{sys},\texttt{params},\texttt{options}),$$

with the input arguments

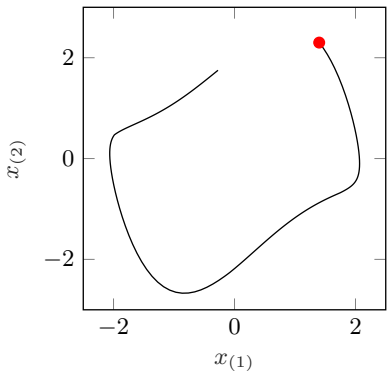
- `sys` dynamic system defined by one of the classes in Sec. 4.2 and 4.3, e.g., `linearSys`, `hybridAutomaton`, etc.
- `params` struct containing the parameter for the simulation
  - `.tStart` initial time $t_0$ (default value 0)
  - `.tFinal` final time $t_f$
  - `.x0` initial point $x_0$
  - `.u` piecewise-constant input signal $u(t)$ specified as a matrix for which the number of rows is identical to the number of system inputs
  - `.p` parameter value $p$ (class `nonlinParamSys` only)
  - `.w` disturbance $w$ (classes `linearSys` and `linearSysDT` only)
  - `.v` sensor noise $v$ (classes `linearSys` and `linearSysDT` only)
  - `.y0guess` guess for a consistent initial algebraic state (class `nonlinDASys` only, see Sec. 4.2.10.1).
  - `.startLoc` index of the initial location (class `hybridAutomaton` and `parallelHybridAutomaton` only)
  - `.finalLoc` index of the final location (class `hybridAutomaton` and `parallelHybridAutomaton` only)
- `options` simulation options for MATLAB's `ode45` function (see `https://de.mathworks.com/help/matlab/ref/ode45.html`).

and the output arguments

- `t` time points of the simulated trajectory
- `x` states of the simulated trajectory
- `ind` index of event function triggered by MATLAB's `ode45` function (see `https://de.mathworks.com/help/matlab/ref/ode45.html`)
- `y` output trajectory (classes `linearSys` and `linearSysDT` only)

Let us demonstrate the operation `simulate` by an example:

```
% nonlinear system
f = @(x,u) [x(2) + u; ...
            (1-x(1)^2)*x(2)-x(1)];

sys = nonlinearSys(f);

% parameter
params.x0 = [1.4;2.3];
params.tFinal = 6;
params.u = [0.1 0 -0.1 0.2];

% simulation
[t,x] = simulate(sys,params);
```




### 4.1.5 simulateRandom

The operation `simulateRandom` simulates a dynamic system for multiple random initial states $x_0 \in \mathcal{X}_0$ and random values for the inputs $u(t) \in \mathcal{U}$ as well as parameters $p \in \mathcal{P}$. The syntax is as follows:

$$\texttt{simRes} = \texttt{simulateRandom}(\texttt{sys}, \texttt{params}, \texttt{options})$$

with input arguments

- `sys` dynamic system defined by one of the classes in Sec. 4.2 and 4.3, e.g., `linearSys`, `hybridAutomaton`, etc.
- `params` struct containing the parameters that define the reachability problem. The parameters are identical to those for the operation `reach` (see Sec. 4.1.1).
- `options` struct containing settings for the random simulation
  - `.points` number of random initial states (positive integer)
  - `.type` sampling method: `'standard'` (default, undefined distribution), `'gaussian'` (Gaussian distribution), `'rrt'` (sampling using *rapidly exploring random trees*)

  depending on the sampling method, there are different additional settings

  `.type = 'standard'`: standard sampling method (undefined distribution)
  - `.fracVert` percentage of initial states randomly drawn from the vertices of the initial set $\mathcal{X}_0$ (value in $[0, 1]$)
  - `.fracInpVert` percentage of input values randomly drawn from the vertices of the input set $\mathcal{U}$ (value in $[0, 1]$)
  - `.nrConstInp` number piecewise-constant input values within the input signal during simulation (integer $\geq 1$)

  `.type = 'rrt'`: sampling using *rapidly-exploring random trees*
  - `.vertSamp` flag specifying whether random initial states, inputs, and parameters are sampled from the vertices of the corresponding sets (0 or 1).
  - `.stretchFac` stretching factor for enlarging the reachable sets during execution of the algorithm (scalar $> 1$).
  - `.R` object of class `reachSet` (see Sec. 7.1) that stores the reachable set for the corresponding reachability problem.

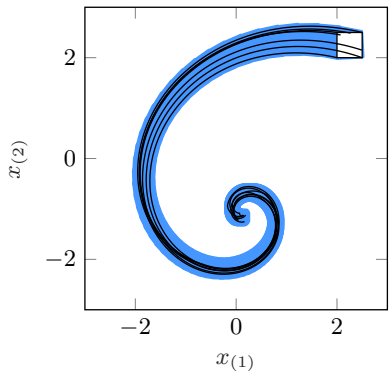
  `.type = 'gaussian'`: sampling from gaussian distribution
  - `.nrConstInp` number piecewise-constant input values within the input signal during simulation (integer $\geq 1$)
  - `.p_conf` probability that sampled value is within the set (value in $(0, 1)$)

and output arguments

- `simRes` object of class `simResult` (see Sec. 7.2) that stores the simulated trajectories.

Let us demonstrate the operation `simulateRandom` by an example:

```
% system dynamics
sys = linearSys([-0.7 -2;2 -0.7],[1;1],[-2;-1]);

% parameter
params.tFinal = 5;
params.R0 = zonotope(interval([2;2],[2.5;2.5]));
params.U = zonotope(interval(-0.1,0.1));

% simulation settings
options.points = 7;
options.fracVert = 0.5;
options.fracInpVert = 1;
options.nrConstInp = 10;

% random simulation
simRes = simulateRandom(sys,params,options);
```




### 4.1.6 verify

The operation `verify` automatically verifies given specifications for the dynamic system and parameters. Currently, this operation is primarily supported for dynamic systems of class `linearSys`.

The syntax for the operation `verify` is:

$$\texttt{res} = \texttt{verify}(\texttt{sys}, \texttt{params}, \texttt{options}, \texttt{spec}),$$

with input arguments

- `sys` dynamic system defined by any of the classes in Sec. 4.2 and 4.3, e.g., `linearSys`, `hybridAutomaton`, etc.
- `params` struct containing the parameter that define the reachability problem
- `options` struct containing algorithm settings for the verification.
- `spec` object of class `specification` (see Sec. 7.3) or class `stl` (see Sec. 7.9), which represents the specifications the system has to fulfill.

and output arguments

- `res` Boolean flag that indicates whether the specifications are satisfied (`res` = 1) or not (`res` = 0).

Let us demonstrate the operation `verify` by an example:

```
% specification
x = stl('x',2);
spec = until(x(2) > 0.4,x(1) < 0,interval(0,2));

% dynamic system
sys = linearSys([0 -1; 1 0],[0; 0]);

% reachability parameters
params.R0 = zonotope(interval([0.5;0.5],[1;1]));

% algorithm settings
options.verifyAlg = 'stl:seidl';
options.taylorTerms = 10;
options.zonotopeOrder = 10;

% verify
verify(sys,params,options,spec)
```

### 4.1.7 observe

The operation `observe` performs guaranteed state estimation to obtain the set of possible states from inputs and outputs. Since measurements are typically obtained at discrete points in time, we only discuss the discrete-time case subsequently. To formalize the problem of set-based state estimation, we introduce the operator to receive the next state as $\chi(x_k, u_k, w_k)$. Our goal is to obtain the set of states $\mathcal{S}_k$ at time step $k$ enclosing the true state from a set of initial states $\mathcal{S}_0 \subset \mathbb{R}^n$, which we define inductively:

$$\mathcal{S}_k = \left\{ x_k = \chi\Big(x_{k-1}, u_{k-1}, w_{k-1}\Big) \middle| x_{k-1} \in \mathcal{S}_{k-1}, w_{k-1} \in \mathcal{W}, v_k \in \mathcal{V}, y_k = Cx_k + v_k \right\}.$$

A reachability problem is a special case, which does not require to check the consistency with the measurement to obtain the reachable set as

$$\mathcal{R}_k = \left\{ x_k = \chi\Big(x_{k-1}, u_{k-1}, w_{k-1}\Big) \middle| x_{k-1} \in \mathcal{S}_{k-1}, w_{k-1} \in \mathcal{W} \right\}.$$

We aim at computing an over-approximation of $\mathcal{S}_k$ that minimizes various cost functions as described in [54, 55]. This goal is pursued differently for the strip-based, set-propagation, and interval observers [54, 55].

The syntax for the operation `observe` is:

$$\texttt{R} = \texttt{observe}(\texttt{sys}, \texttt{params}, \texttt{options}),$$

with input arguments

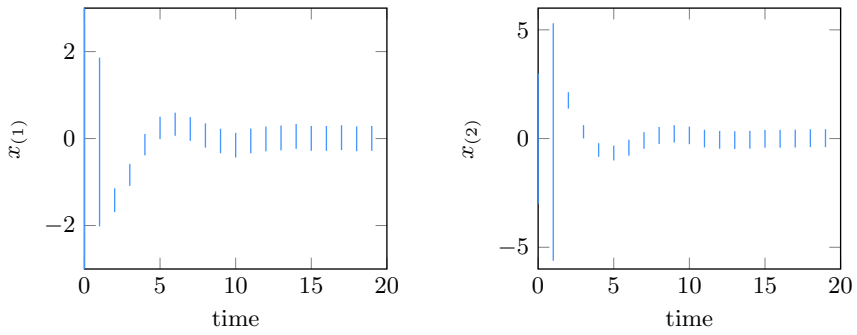
- `sys` dynamic system defined by one of the classes `linearSysDT` (see Sec. 4.2.3) or `nonlinearSysDT` (see Sec. 4.2.8)
- `params` struct containing the parameter that define the observation problem
  - `.tStart` initial time $t_0$ (default value 0)
  - `.tFinal` final time $t_f$
  - `.R0` initial set $\mathcal{X}_0$ specified by one of the set representations in Sec. 2.2.1
  - `.W` disturbance $\mathcal{W}$ specified as an object of class `zonotope` (see Sec. 2.2.1.1) or `ellipsoid` (see Sec. 2.2.1.3)
  - `.V` set of sensor noises $\mathcal{V}$ specified as an object of class `zonotope` (see Sec. 2.2.1.1) or `ellipsoid` (see Sec. 2.2.1.3)
  - `.u` time-dependent input $u(t)$ to the system, specified as a matrix for which the number of columns is identical to the number of measurements
  - `.y` time-dependent output $y(t)$ to the system, specified as a matrix for which the number of columns is identical to the number of measurements
- `options` struct containing algorithm settings for set-based observation. Since the settings are different for each type of dynamic system, they are documented in Sec. 4.2 and Sec. 4.3.

and output arguments

- `R` object of class `reachSet` (see Sec. 7.1) that stores the observed set $\mathcal{R}(t_i)$ for all time points.

Let us demonstrate the operation `observe` by an example:

```
%% Parameters
params.tFinal = 20; %final time
params.R0 = zonotope(zeros(2,1),3*eye(2)); %initial set
params.V = 0.2*zonotope([0,1]); % sensor noise set
params.W = 0.02*[-6; 1]*zonotope([0,1]); % disturbance set
params.u = zeros(2,1); % input vector
params.y = [0.79, 5.00, 4.35, 1.86, -0.11, -1.13, -1.17, -0.76, ...
    -0.12, 0.72, 0.29, 0.19, 0.09, -0.21, 0.05, -0.00, -0.16, 0.01, ...
    -0.08, 0.13]; %measurement vector


%% Algorithmic Settings
options.zonotopeOrder = 20; % zonotope order
options.timeStep = 1; % step size
options.alg = 'FRad-C'; % observer approach

%% System Dynamics
reactor = linearSysDT('reactor',[0 -0.5; 1 1], 1, zeros(2,1), [-2 1], options.timeStep);

% observe
EstSet = observe(reactor,params,options);
```




### 4.1.8 computeGO

The operation `computeGO` computes the parameters of a general output (GO) model, which provides a linear approximation of the output of the system along a reference trajectory, using a first-order Taylor series expansion [6]. For the reference trajectory defined by the initial state $\bar{x}[1]$ and the inputs $\bar{u}[i]$, $i = 1, ..., k$, the output of the GO model for the state $x[1]$ and the inputs $u[i]$ is given as

$$y_{GO}[k] = \bar{y}[k] + \bar{C}[k](x[1] - \bar{x}[1]) + \sum_{i=1}^{k} \bar{D}_i[k](u[i] - \bar{u}[i]), \tag{29}$$

We can extend the GO model to also provide a linear approximation of the system state, which is given as:

$$x_{GO}[k+1] = \bar{x}[k+1] + \bar{A}[k](x[1] - \bar{x}[1]) + \sum_{i=1}^{k} \bar{B}_i[k](u[i] - \bar{u}[i]). \tag{30}$$

With the set $\mathcal{L}[i] = \mathcal{L}_x[i] \times \mathcal{L}_y[i]$, which contains higher-order Taylor terms and the Lagrange remainder for the Taylor series expansion at time step $i$, we can compute the reachable set of states $\mathcal{X}$ and the reachable set of outputs $\mathcal{Y}$ of the system for the uncertain initial state

$x[1] \in \mathcal{X}[1]$ and uncertain inputs $u[i] \in \mathcal{U}[i]$ with the GO model as

$$\mathcal{X}[k+1] = \bar{x}[k+1] \oplus \bar{A}[k](\mathcal{X}[1] - \bar{x}[1]) \oplus \bigoplus_{i=1}^{k} \bar{B}_i[k](\mathcal{U}[i] - \bar{u}[i]) \oplus \bigoplus_{i=1}^{k} \bar{F}_i[k]\mathcal{L}[i] \tag{31}$$

$$\mathcal{Y}[k] = \bar{y}[k] \oplus \bar{C}[k](\mathcal{X}[1] - \bar{x}[1]) \oplus \bigoplus_{i=1}^{k} \bar{D}_i[k](\mathcal{U}[i] - \bar{u}[i]) \oplus \bigoplus_{i=1}^{k} \bar{E}_i[k]\mathcal{L}[i]. \tag{32}$$

The syntax for the operation `computeGO` is:

$$\texttt{p\_GO} = \texttt{computeGO}(\texttt{sys}, \texttt{x0\_ref}, \texttt{u\_ref}, \texttt{n\_k})$$

with input arguments

- `sys` dynamic system defined by one of the classes `linearSysDT` (see Sec. 4.2.3), `linearARX` (see Sec. 4.2.5), `nonlinearSysDT` (see Sec. 4.2.8), or `nonlinearARX` (see Sec. 4.2.9).
- `x0_ref` initial state of the reference trajectory, equivalent to $\bar{x}[1]$ in (30) and (29).
- `u_ref` inputs for the reference trajectory, equivalent to $\bar{u}[\cdot]$ in (30) and (29).
- `n_k` time horizon.

and output arguments

- `p_GO` parameters of the GO model with $k = 1, ..., n\_k$ and $i = 1, ..., k$.
  - `.A{k}` matrix that describes the influence of the initial state $x[1]$ on the state $x[k+1]$, equivalent to $\bar{A}[k]$ in (30) and (31).
  - `.B{k,i}` matrix that describes the influence of the input $u[i]$ on the state $x[k+1]$, equivalent to $\bar{B}_i[k]$ in (30) and (31).
  - `.F{k,i}` matrix that describes the influence of the linearization error $l[i]$ on the state $x[k+1]$, equivalent to $\bar{F}_i[k]$ in (31).
  - `.C{k}` matrix that describes the influence of the initial state $x[1]$ on the output $y[k]$, equivalent to $\bar{C}[k]$ in (29) and (32).
  - `.D{k,i}` matrix that describes the influence of the input $u[i]$ on the output $y[k]$, equivalent to $\bar{D}_i[k]$ in (29) and (32).
  - `.E{k,i}` matrix that describes the influence of the linearization error $l[i]$ on the output $y[k]$, equivalent to $\bar{E}_i[k]$ in (32).
  - `.x(:,k)` reference state, equivalent to $\bar{x}[k]$.
  - `.u(:,k)` reference input, equivalent to $\bar{u}[k]$.
  - `.y(:,k)` reference output, equivalent to $\bar{y}[k]$ .

Let us demonstrate the operation `computeGO` by an example:

```
%% Reference trajectory
x0_ref = [0; 0]; %initial state of reference trajectory
u_ref = zeros(2,10); %inputs for reference trajectory
n_k = 5; %time horizon

%% System Dynamics
reactor = linearSysDT('reactor',[0 -0.5; 1 1],  eye(2), zeros(2,1), [-2 1], 0.1);

% conformance synthesis
p_GO = computeGO(reactor,x0_ref,u_ref,n_k);
```

### 4.1.9 isconform

The operation `isconform` checks reachset conformance [56, 57]. A model is essentially reachset conformant if the measurements of the real system are contained in the set of reachable outputs of the model. Instead of checking reachset conformance with respect to a real system, one can also check reachset conformance with respect to a high-fidelity model.

Formally, reachset conformance checks whether for all times $t$ and input trajectories $u(\cdot) \in \mathcal{U}$ ($\mathcal{U}$ is the set of input trajectories), the set of reachable outputs $Reach_t(S_A, u(\cdot))$ of the abstract system $S_A$ contains all possible measurements $Reach_t(S_I, u(\cdot))$ of the implementation $S_I$:

$$S_I \,\texttt{conf}_R\, S_A \quad \Longleftrightarrow \quad \forall t,\, \forall u(\cdot) \in \mathcal{U} : Reach_t(S_I, u(\cdot)) \subseteq Reach_t(S_A, u(\cdot)).$$

In [58, Thm. 1.], it is shown that reachset conformance is necessary and sufficient to transfer safety properties, which are the predominant properties for certifying cyber-physical systems. In addition, properties that can be formalized using reachset temporal logic [59] can be transferred. Reachset conformance is a weaker conformance notion than trace conformance, i.e., if $S_A$ is a trace conformant model of $S_I$, it is also a reachset conformant model of $S_I$ [58, Thm. 2].

Reachset conformance can only be proven between models because one obviously cannot compute the reachable set of a real system. For real systems, one resorts to checking whether the reachable output of the abstract model $S_A$ contains all measurements of the implementation $S_I$. Even when $S_I$ is a model, one often resorts to using simulation results rather than reachability analysis due to the computational complexity of computing reachable sets. We call this approach reachset conformance checking, because we can only check the containment of samples rather than proving reachset conformance. To the best knowledge of the author, reachset conformance checking was first presented in [60]. However, the term *reachset conformance* is not used in that work and was introduced in [53] together with a formalization of reachset conformance checking.

The syntax for the operation `isconform` is:

$$[\texttt{res}, \texttt{R}, \texttt{simRes}, \texttt{unifiedOutputs}] = \texttt{isconform}(\texttt{sys}, \texttt{params}, \texttt{options}, \texttt{type})$$

with input arguments

- `sys` dynamic system defined by one of the classes `linearSysDT` (see Sec. 4.2.3) or `nonlinearSysDT` (see Sec. 4.2.8)
- `params` struct containing the parameter that define the conformance problem
  - `.tStart` initial time $t_0$ (default value 0).
  - `.tFinal` final time $t_f$.
  - `.R0` uncertainty set $\tilde{\mathcal{X}}_0$ relative to the initial state, specified by one of the set representations in Sec. 2.2.1.
  - `.W` set of disturbances $\mathcal{W}$ specified as an object of class `zonotope` (see Sec. 2.2.1.1).
  - `.V` set of sensor noises $\mathcal{V}$ specified as an object of class `zonotope` (see Sec. 2.2.1.1).
  - `.testSuite` cell array of testCase objects, which contain input trajectories and the corresponding output trajectories of the implementation system $S_I$.
- `options` struct containing algorithm settings for conformance checking. Since the settings are different for each type of dynamic system, they are documented in Sec. 4.2 and Sec. 4.3.
- `type` `'RRT'`, `'BF'` or `'dyn'`; can be omitted for `'dyn'`.

and output arguments

- `res` result: true/false for conformance checking.
- `R` object of class `reachSet` (see Sec. 7.1) that stores the observed set $\mathcal{R}(t_i)$ for all time points.
- `simRes` object of class `simResult` (see Sec. 7.2) that stores simulated trajectories in case a white-box model is used for reachset conformance.
- `unifiedOutputs` Methods for linear systems unify outputs using the superposition principle, which are collected in this matrix.

Let us demonstrate the operation `isconform` by an example:

```
%% Parameters
dt = 1; %sampling time
params.tFinal = 5; %final time
params.R0 = zonotope(zeros(2,1),eye(2)); %initial set
params.V = zonotope([0,1]); % sensor noise set
params.W = [-6; 1]*zonotope([0,1]); % disturbance set
y = [0.79; 5.00; 4.35; 1.86; -0.11; -1.13]; %measurement vector
delta = [0.1; 0.2; 0.1; 0; -0.1; -0.2]; %deviation of measurement vector
params.testSuite{1} = testCase(y, zeros(6,2), [1,1], dt); %test case 1
params.testSuite{2} = testCase(y + delta, zeros(6,2), [1,1], dt); %test case 2
params.testSuite{3} = testCase(y - delta, zeros(6,2), [1,1], dt); %test case 3


%% Algorithmic Settings
options = struct; % use default settings

%% System Dynamics
reactor = linearSysDT('reactor',[0 -0.5; 1 1], eye(2), zeros(2,1), [-2 1], dt);

% conformance synthesis
res = isconform(reactor,params,options);
```

### 4.1.10 conform

The operation `conform` identifies a reachset-conformant model (see Sec. 4.1.9 for a detailed explanation of reachset conformance). Depending on our prior knowledge about the model, we can solely identify the uncertainty sets, which make the model reachset-conformant, (termed white-box identification) or we can also identify unknown model parameters (gray-box identification) or the whole model dynamics (black-box identification) [5, 6]. Reachset conformance has been established in numerous applications, such as safe human-robot co-existence [61], safe robot manipulators [62], force control [63], and analog circuits [64].

The syntax for the operation `conform` is:

$$[\texttt{params}, \texttt{results}] = \texttt{conform}(\texttt{sys}, \texttt{params}, \texttt{options}, \texttt{type})$$

with input arguments

Table 7: Conformance identification algorithms.

| Algorithm | Description | Reference |
|---|---|---|
| `white` | identification of the scaling factors and (for linear systems) center vectors of the uncertainty sets, which make a given model reachset-conformant, with linear programming | [5] |
| `graySim` | simultaneous identification of the uncertainty sets and unknown model parameters with nonlinear programming. | [6] |
| `graySeq` | sequential identification of unknown model parameters and the uncertainty sets with nonlinear programming. | [6] |
| `grayLS` | sequential identification of unknown model parameters and the uncertainty sets with nonlinear programming using least squares cost function. | [6] |
| `blackCGP` | identification of a reachset-conformant model using conformant genetic programming. | [6] |
| `blackGP` | identification of a reachset-conformant model using standard genetic programming | [6] |
| `RRT` | increase additive uncertainty sets until all trajectories, generated with rapidly exploring random trees (RRTs), are contained in the reachable set of the model. | [60] |

- `sys` dynamic system defined by one of the classes `linearSysDT` (see Sec. 4.2.3), `linearARX` (see Sec. 4.2.5), `nonlinearSysDT` (see Sec. 4.2.8), or `nonlinearARX` (see Sec. 4.2.9).
- `params` struct containing the parameter that define the conformance problem.
  - `.tStart` initial time $t_0$ (default value 0).
  - `.tFinal` final time $t_f$.
  - `.R0` initial uncertainty set $\mathcal{X}_0$ relative to the initial state, specified as an object of class `zonotope` (see Sec. 2.2.1.1).
  - `.U` initial uncertainty set $\mathcal{U}$ relative to the input, specified as an object of class `zonotope` (see Sec. 2.2.1.1).
  - `.testSuite` cell array of testCase objects, which contain input trajectories and the corresponding output trajectories of the implementation system $S_I$.

- `options` struct containing algorithm settings for conformance synthesis. For the white-, gray-, and black-box identification algorithms, we can specify:
  - `.cs.cost` cost function for reachset conformance synthesis, currently the interval norm (`interval`) and the Frobenius norm (`Frob`) are supported (see [56]).
  - `.cs.w` weighting vector for the norm at different times for reachset conformance synthesis, the default value is a vector of ones.
  - `.cs.P` weighting matrix for the Frobenius norm, the default value is the identity matrix.
  - `.cs.constraints` type of containment constraints, currently halfspace constraints (`half`) and generator constraints (`gen`) are supported (see [5]).
  - `.cs.robustnessMargin` robustness value for enforcing the containment constraints, the default value is $1e-9$.
  - `.cs.derivRecomputation` boolean, which can enforce the Jacobian and Hessian recomputation (required for linearization) at each nonlinear programming iteration.
  - `.cs.a_min` lower limit for the identified scaling factors, default is 0.
  - `.cs.a_max` upper limit for the identified scaling factors, default is $\infty$.
  - `.cs.cp_lim` upper limit for the absolute value of the identified center vectors and parameters, default is $\infty$.
  - `.cs.verbose` boolean, which can suppress displaying output.

  For the gray-box algorithms, `options` can also contain:
  - `.cs.p0` initial estimate for the parameters.
  - `.cs.set_p` function handle, which return the system object and the uncertainty sets for a given parameter vector (default: parameter vector contains the center vectors of the uncertainty sets).
  - `.cs.p_min` lower limits for the parameter vector, default is `options.cs.p0`−`options.cs.cp_lim`.
  - `.cs.p_max` upper limits for the parameter vector, default is `options.cs.p0`+`options.cs.cp_lim`.
  - `.cs.cp_lim` upper limit for the absolute value of the identified center vectors and parameters, default is $\infty$.
  - `.cs.timeout` time in $s$ after which the nonlinear programming solver is forced to stop.

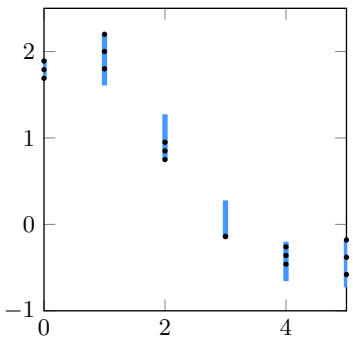
  For the black-box algorithms, we can also specify a large number of settings for the genetic programming approach. A list can be found in the file `config_contDynamics_conform_black`.
- `type` algorithm type (see Tab. 7); can be omitted for `white`.

and output arguments

- `params` struct containing the updated uncertainty sets
- `results` struct with the optimization results, which depend on the algorithm type and can be:
  - `.R` object of class `reachSet` (see Sec. 7.1) that stores the observed set $\mathcal{R}(t_i)$ for all time points if the algorithm is `RRT`.
  - `.simRes` object of class `simResult` (see Sec. 7.2) that stores simulated trajectories if the algorithm is `RRT`.
  - `.fval` final optimization cost.
  - `.p` optimized parameters.
  - `.sys` optimized dynamic system.
  - `.unifiedOutputs` Methods for linear systems unify outputs using the superposition principle, which are collected in this matrix.

Let us demonstrate the operation `conform` by an example (the plot shows the reachable output set of the conformant model and the measured outputs):

```
%% Parameters
dt = 1; %sampling time
params.tFinal = 5; %final time
params.R0 = zonotope(zeros(2,1),eye(2)); %initial set
params.V = zonotope([0,1]); % sensor noise set
params.W = [-6; 1]*zonotope([0,1]); % disturbance set
y = [0.79; 5.00; 4.35; 1.86; -0.11; -1.13]; %measurement vector
delta = [0.1; 0.2; 0.1; 0; -0.1; -0.2]; %deviation of measurement vector
params.testSuite{1} = testCase(y, zeros(6,2), [1,1], dt); %test case 1
params.testSuite{2} = testCase(y + delta, zeros(6,2), [1,1], dt); %test case 2
params.testSuite{3} = testCase(y - delta, zeros(6,2), [1,1], dt); %test case 3


%% Algorithmic Settings
options.cs.cost = 'interval'; % interval norm
options.cs.constraints = 'gen'; % generator constraints

%% System Dynamics
reactor = linearSysDT('reactor',[0 -0.5; 1 1],  eye(2), zeros(2,1), [-2 1], dt);

% conformance synthesis
paramsConform = conform(reactor,params,options);
```




### 4.1.11 cora2spaceex

The operation `cora2spaceex` convertes a dynamical system represented as a CORA object to a SpaceEx model [65]. The syntax is as follows:

`cora2spaceex(sys, fileName)`,

with the input arguments

- `sys` dynamic system represented as an object of class `linearSys` (see Sec. 4.2.1), `nonlinearSys` (see Sec. 4.2.6), or `hybridAutomaton` (see Sec. 4.3.1).
- `fileName` name of the converted SpaceEx file.

Let us demonstrate the operation `cora2spaceex` by an example:

```
% nonlinear system
f = @(x,u) [x(2); ...
            (1-x(1)^2)*x(2)-x(1)];

sys = nonlinearSys(f);

% convert to SpaceEx model
cora2spaceex(sys,'vanDerPol');
```

```
<?xml version="1.0" encoding="utf-8"?>
<sspaceex math="spaceex" version="2.0">
  <component id="model">
    <param name="x1" type="real"/>
    <param name="x2" type="real"/>
    <location id="1">
      <invariant/>
      <flow>
        x1' == x2 &amp;
        x2' == - x1 - x2*(x1^2 - 1)
      </flow>
    </location>
  </component>
</sspaceex>
```

## 4.2 Continuous Dynamics

This section introduces various classes to represent different types of continuous dynamics. CORA supports the following continuous dynamics:

- Linear systems (Sec. 4.2.1)
- Linear systems with uncertain parameters (Sec. 4.2.2)
- Linear discrete-time systems (Sec. 4.2.3)
- Linear probabilistic systems (Sec. 4.2.4)
- Linear ARX models (Sec. 4.2.5)
- Nonlinear systems (Sec. 4.2.6)
- Nonlinear systems with uncertain parameters (Sec. 4.2.7)
- Nonlinear discrete-time systems (Sec. 4.2.8)
- Nonlinear ARX models (Sec. 4.2.9)
- Nonlinear differential-algebraic systems (Sec. 4.2.10)
- Neural network controlled systems (Sec. 4.2.11)

Each class for continuous dynamics inherits from the parent class `contDynamics` (see Fig. 1). Next, we explain all classes in detail.

### 4.2.1 Linear Systems

The first system dynamics we consider are linear systems of the form

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) + c + w(t), \\ y(t) &= Cx(t) + Du(t) + k + v(t), \end{aligned} \tag{33}$$

where $x(t) \in \mathbb{R}^n$ is the system state, $u(t) \in \mathbb{R}^m$ is the system input, $w(t) \in \mathbb{R}^n$ is the disturbance, $y(t) \in \mathbb{R}^p$ is the system output, $v(t) \in \mathbb{R}^p$ is the sensor noise, and $A \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{n \times m}$, $c \in \mathbb{R}^n$,

$C \in \mathbb{R}^{p \times n}$, $D \in \mathbb{R}^{p \times m}$, $k \in \mathbb{R}^p$. Linear systems are implemented by the class `linearSys`. An object of class `linearSys` can be constructed as follows:

$$\begin{aligned}
&\texttt{sys} = \texttt{linearSys}(A, B),\\
&\texttt{sys} = \texttt{linearSys}(A, B, c, C, D, k),\\
&\texttt{sys} = \texttt{linearSys}(\texttt{name}, A, B),\\
&\texttt{sys} = \texttt{linearSys}(\texttt{name}, A, B, c, C, D, k),
\end{aligned}$$

where `name` is a string specifying the name of the system and $A, B, c, C, D, k$ are defined as in (33). Let us demonstrate the class `linearSys` by an example:

$$\begin{aligned}
\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix} &= \begin{bmatrix}-2 & 0\\ 1 & -3\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} + \begin{bmatrix}1\\ 1\end{bmatrix} u\\
y &= \begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix}
\end{aligned}$$

```
% system matrices
A = [-2 0; 1 -3];
B = [1; 1];
C = [1 0];

% linear system
sys = linearSys(A,B,[],C);
```

#### 4.2.1.1 Operation `reach`

There exist several different algorithms for computing the reachable set of a linear system. The algorithms implemented in CORA are listed in Tab. 8. We recommend to use the adaptive algorithm (`options.linAlg = 'adaptive'`) since it is fully automatic and does not require any manual parameter tuning. A visualization of the basic steps that are applied to calculate the reachable set for a linear system is shown in Fig. 9: First, the reachable set $\mathcal{R}_h^d = e^{A\Delta t}\mathcal{X}_0$ for the next point in time is computed by propagating the initial set $\mathcal{X}_0$ with the matrix exponential $e^{A\Delta t}$. In the second step, the convex hull is computed. To account for the curvature of trajectories, the set resulting from the convex hull is bloated by an error term in the third step, which yields a tight enclosure of the reachable set $\mathcal{R}^d(\tau_0)$ for the time interval $\tau_0$.

Table 8: Reachability algorithms for linear systems.

| Algorithm | Description | Reference |
|---|---|---|
| `standard` | standard algorithm | [32] |
| `wrapping-free` | avoid wrapping effect | [66] |
| `fromStart` | propagation from start | [14] |
| `decomp` | block decomposition (high-dim. systems) | [67] |
| `krylov` | Krylov subspace method (high-dim. systems) | [68][17] |
| `adap` | determine near-optimal settings automatically | [69] |

The settings for reachability analysis are specified as fields of the struct `options` (see Sec. 4.1.1). For linear systems, the following settings are available:

- `.linAlg` — string specifying the reachability algorithm that is used (see Tab. 8). The default value is `'standard'`.
- `.timeStep` — time step size required for all algorithms except for `'adaptive'`.

[17] Requires *Multiple Precision Toolbox*:
`https://www.mathworks.com/matlabcentral/fileexchange/6446-multiple-precision-toolbox-for-matlab`

- `.taylorTerms` — number of Taylor terms for the computation of the exponential matrix $e^{A\Delta t}$ (see [33, Eq. (3.2)]). Required for all algorithms except for `'adaptive'`.
- `.zonotopeOrder` — upper bound for the zonotope order $\rho$ (see Sec. 2.2.1.1). Required for all algorithms except for `'adaptive'`.
- `.reductionTechnique` — string specifying the method used to reduce the zonotope order (see Tab. 4). The default value is `'girard'`.
- `.partition` — array defining the range of dimensions each block covers. All blocks together make up the linear system (algorithm `'decomp'` only).
- `.krylovError` — upper bound of Krylov error as defined in [68, eq. (3)] (algorithm `'krylov'` only)
- `.krylovStep` — step size to increase the dimension of the Krylov subspace $\xi$ as defined in [68, Sec. II.A] until the Krylov error is below the upper bound defined by `.krylovError` (algorithm `'krylov'` only)
- `.error` — upper bound for the error containing over-approximative terms as defined in [69] (algorithm `'adaptive'` only). The default value is set to one hundreth of the longest edge of the interval over-approximation of the initial set.

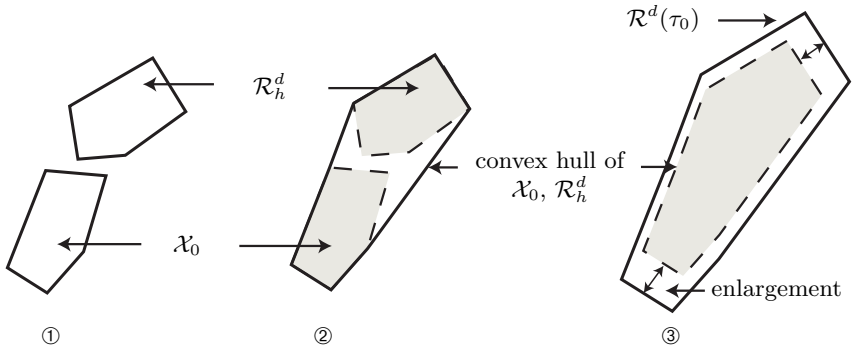



Figure 9: Steps for the computation of an over-approximation of the reachable set for a linear system.

#### 4.2.1.2 Operation `reachInner`

To obtain an inner-approximation, we compute the reachable set for piecewise-constant uncertain inputs, which is an inner-approximation of the reachable set with uncertain inputs that can vary arbitrarily over time. Furthermore, we compute an inner-approximation of the original zonotope when we reduce the zonotope order.

The settings for reachability analysis are specified as fields of the struct `options` (see Sec. 4.1.2). For linear systems, the following settings are available:

- `.timeStep` time step size $\Delta t$.
- `.zonotopeOrder` upper bound for the zonotope order $\rho$ (see Sec. 2.2.1.1).
- `.reductionTechnique` string specifying the method used to reduce the zonotope order in an under-approximative way. The available methods are `'sum'`, `'linProg'`, `'scale'`. The default value is `'sum'`.

### 4.2.2 Linear Systems with Uncertain Parameters

This class extends linear systems by uncertain parameters. We provide two implementations, one for uncertain parameters that are fixed over time and one for parameters that can arbitrarily vary over time. For the case with fixed parameters, a linear parametric system is defined as

$$\dot{x}(t) = A(p)\ x(t) + B(p)\ u(t), \quad p \in \mathcal{P},$$

which can be equivalently formulated as

$$\begin{aligned} &\dot{x}(t) = Ax(t) + Bu(t), \quad A \in \mathcal{A},\ B \in \mathcal{B}, \\ &\text{with}\ \ \mathcal{A} = \{A(p) \mid p \in \mathcal{P}\},\ \mathcal{B} = \{B(p) \mid p \in \mathcal{P}\}, \end{aligned} \tag{34}$$

where $x(t) \in \mathbb{R}^n$ is the system state, $u(t) \in \mathbb{R}^m$ is the system input, $p \in \mathbb{R}^p$ is the parameter vector, and $\mathcal{P} \subset \mathbb{R}^p$ is the set of parameters. For the case with fixed parameters, a linear parametric system is defined as

$$\dot{x}(t) = A(t)\ x(t) + B(t)\ u(t), \quad A(t) \in \mathcal{A},\ B(t) \in \mathcal{B},$$

where $\mathcal{A}$ and $\mathcal{B}$ are defined as in (34). Linear parametric systems are implemented by the class `linParamSys`. An object of class `linParamSys` can be constructed as follows:

$$\begin{aligned} &\texttt{sys} = \texttt{linParamSys}(\mathcal{A}, \mathcal{B}), \\ &\texttt{sys} = \texttt{linParamSys}(\mathcal{A}, \mathcal{B}, \texttt{type}), \\ &\texttt{sys} = \texttt{linParamSys}(\texttt{name}, \mathcal{A}, \mathcal{B}), \\ &\texttt{sys} = \texttt{linParamSys}(\texttt{name}, \mathcal{A}, \mathcal{B}, \texttt{type}), \end{aligned}$$

where `name` is a string specifying the name of the system, $\mathcal{A}, \mathcal{B}$ are defined as in (34), and `type` is a string specifying whether the parameters are constant over time (`'constParam'`) or time-varying (`'varParam'`). The default value for `type` is `'constParam'`. The matrix sets $\mathcal{A}$ and $\mathcal{B}$ can be represented by any of the matrix set representations introduced in Sec. 3. Let us demonstrate the class `linParamSys` by an example:

$$\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix} = \begin{bmatrix}-2 & 0\\ [1,2] & -3\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} + \begin{bmatrix}1\\ 1\end{bmatrix} u$$

```
% system matrices
Ac = [-2 0; 1.5 -3];
Aw = [0 0; 0.5 0];
A = intervalMatrix(Ac,Aw);

B = [1; 1];

% linear parametric system
sys = linParamSys(A,B,'varParam');
```

An alternative for fixed parameters is to define each parameter as a state variable $\tilde{x}_i$ with the trivial dynamics $\dot{\tilde{x}}_i = 0$. For time-varying parameters, one can specify the parameter as an uncertain input. In both cases, the result is a nonlinear system that can be handled as described in Sec. 4.2.6. The question of whether to compute the solution with the dedicated approach presented in this section or with the approach for nonlinear systems has not yet been thoroughly investigated.

#### 4.2.2.1 Operation `reach`

Reachability analysis for linear parametric systems is very similar to reachability analysis of linear systems with known parameters. The main difference is that we have to take into account an uncertain state matrix $\mathcal{A}$ and an uncertain input matrix $\mathcal{B}$. We apply the algorithm from [70] to calculate the reachable set of linear parametric systems.

The settings for reachability analysis are specified as fields of the struct `options` (see Sec. 4.1.1). For linear systems, the following settings are available:

- `.timeStep` — time step size for one reachability time step.
- `.taylorTerms` — number of Taylor terms for the computation of the exponential matrix $e^{\mathcal{A}\Delta t}$ (see [33, Theorem 3.2]).
- `.zonotopeOrder` — upper bound for the zonotope order $\rho$ (see Sec. 2.2.1.1).
- `.reductionTechnique` — string specifying the method used to reduce the zonotope order (see Tab. 4). The default value is `'girard'`.
- `.intermediateTerms` — upper bound for the zonotope order $\rho$ (see Sec. 2.2.1.1) in internal computations of the algorithm.
- `.compTimePoint` — flag specifying whether the reachable sets should be computed for points in time (`compTimePoint = 1`) or not (`compTimePoint = 0`). The default value is `0`.

### 4.2.3 Linear Discrete-Time Systems

In addition to continuous-time linear systems, CORA also supports discrete-time linear systems defined as

$$\begin{aligned} x[i+1] &= Ax[i] + Bu[i] + c + w[i],\\ y[i] &= Cx[i] + Du[i] + k + v[i], \end{aligned} \tag{35}$$

where $x[i] \in \mathbb{R}^n$ is the system state, $u[i] \in \mathbb{R}^m$ is the system input, $w[i] \in \mathbb{R}^n$ is the disturbance, $y[i] \in \mathbb{R}^p$ is the system output, $v[i] \in \mathbb{R}^p$ is the sensor noise, and $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, $c \in \mathbb{R}^n$, $C \in \mathbb{R}^{p\times n}$, $D \in \mathbb{R}^{p\times m}$, $k \in \mathbb{R}^p$. Discrete-time linear systems are implemented by the class

`linearSysDT`. An object of class `linearSysDT` can be constructed as follows:

$$\begin{aligned}
&\texttt{sys} = \texttt{linearSysDT}(A, B, \Delta t),\\
&\texttt{sys} = \texttt{linearSysDT}(A, B, c, C, D, k, \Delta t),\\
&\texttt{sys} = \texttt{linearSysDT}(\texttt{name}, A, B, \Delta t),\\
&\texttt{sys} = \texttt{linearSysDT}(\texttt{name}, A, B, c, C, D, k, \Delta t),
\end{aligned}$$

where `name` is a string specifying the name of the system, $A, B, c, C, D, k$ are defined as in (35), and $\Delta t$ is the sampling time specifying the time difference between $x[i+1]$ and $x[i]$.

Let us demonstrate the class `linearSysDT` by an example:

$$\begin{bmatrix} x_1[i+1] \\ x_2[i+1] \end{bmatrix} = \begin{bmatrix} -0.4 & 0.6 \\ 0.6 & -0.4 \end{bmatrix} \begin{bmatrix} x_1[i] \\ x_2[i] \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u[i]$$
$$y[i] = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1[i] \\ x_2[i] \end{bmatrix}$$

```
% system matrices
A = [-0.4 0.6; 0.6 -0.4];
B = [0; 1];
C = [1 0];

% sampling time
dt = 0.4;

% linear discrete-time system
sys = linearSysDT(A,B,[],C,dt);
```

#### 4.2.3.1 Operation `reach`

The reachable set for a linear discrete-time system can be computed by set-based evaluation of (35). After each time step, the zonotope order of the reachable set is reduced to a user-specified order.

The settings for reachability analysis are specified as fields of the struct `options` (see Sec. 4.1.1). For linear discrete-time systems, the following settings are available:

- `.zonotopeOrder` upper bound for the zonotope order $\rho$ (see Sec. 2.2.1.1).
- `.reductionTechnique` string specifying the method used to reduce the zonotope order (see Tab. 4). The default value is `'girard'`.

#### 4.2.3.2 Operation `observe`

The current list of observers for discrete-time linear systems implemented in CORA is shown in Tab. 10. The implemented observers are categorized as *strip-based observers*, *set-propagation observers*, and *interval observers* according to [54,55]. While some reachability analysis approaches are agnostic with respect to the set representation, most approaches for set-based observers are specifically designed for a specific set representation.

Because strip-based observers can only finish their computation of the estimated set after the measurement, their result is always delayed. When a set-propagation observer or interval observer is real-time capable, the estimated set is obtained ahead of time. This issue can be fixed for strip-based observers when additionally computing a one-step prediction and use this set as the initial set as shown in [71, Sec. III]. For this reason, we list these algorithms as not ready for control in Tab. 10.

The settings for set-based estimation are specified as fields of the struct `options` (see Sec. 4.1.1). For linear discrete-time systems, the following settings are available:

Table 10: Algorithms for set-based estimation for discrete-time linear systems.

| Technique | Set representation | Ready for control | Supported Reference |
|---|---|---|---|
| strip-based observers | | | |
| VolMin-A | zonotope | ✗ | [72] |
| VolMin-B | zonotope | ✗ | [73] |
| FRad-A | zonotope | ✗ | [72] |
| FRad-B | zonotope | ✗ | [74] |
| PRad-A | zonotope | ✗ | [75] |
| PRad-B | zonotope | ✗ | [76] |
| PRad-C | zonotope | ✗ | [77] |
| PRad-D | zonotope | ✗ | [78] |
| CZN-A | constr. zono. | ✗ | [34] |
| CZN-B | constr. zono. | ✗ | [79] |
| ESO-A | ellipsoid | ✗ | [80, 81] |
| ESO-B | ellipsoid | ✗ | [81] |
| set-propagation observers | | | |
| FRad-C | zonotope | ✓ | [82] |
| PRad-E | zonotope | ✓ | [83] |
| Nom-G | zonotope | ✓ | [74] |
| ESO-C | ellipsoid | ✓ | [84] |
| ESO-D | ellipsoid | ✓ | [85] |
| interval observer | | | |
| Hinf-G | zonotope | ✓ | [86] |

– `.alg` string specifying the algorithm that is used (see Tab. 10).

– `.zonotopeOrder` upper bound for the zonotope order $\rho$ (see Sec. 2.2.1.1).

– `.reductionTechnique` string specifying the method used to reduce the zonotope order (see Tab. 4). The default value is `'girard'`.

#### 4.2.3.3 Operation `isconform`

The operator `isconform` checks reachset conformance. The list of currently available algorithms is listed in Tab. 11 (the algorithm `RRT` is not recommended for linear systems, while `BF` is mostly used for unit testing).

Table 11: Conformance algorithms for discrete-time linear systems.

| Algorithm | Description | Reference |
|---|---|---|
| `RRT` | checks conformance using RRTs | [60] |
| `BF` | brute-force conformance check | [53] |
| `dyn` | conformance check for linear systems | [56, 87] |

The settings for reachset conformance checking are as for reachability analysis and rapidly-exploring random trees (see Sec. 4.1.1).

### 4.2.4 Linear Probabilistic Systems

In contrast to all other systems, we consider stochastic properties in the class `linProbSys`. The system under consideration is defined by the following linear stochastic differential equation (SDE), which is also known as the multivariate Ornstein-Uhlenbeck process [88]:

$$\begin{aligned} &\dot{x} = Ax(t) + u(t) + C\xi(t), \\ &x(0) \in \mathbb{R}^n,\ u(t) \in \mathcal{U} \subset \mathbb{R}^n,\ \xi \in \mathbb{R}^m, \end{aligned} \tag{36}$$

where $A$ and $C$ are matrices of proper dimension and $A$ has full rank. There are two kinds of inputs: the first input $u$ is Lipschitz continuous and can take any value in $\mathcal{U} \subset \mathbb{R}^n$ for which no probability distribution is known. The second input $\xi \in \mathbb{R}^m$ is white Gaussian noise. The combination of both inputs can be seen as a white Gaussian noise input, where the mean value is unknown within the set $\mathcal{U}$.

In contrast to the other system classes, we compute enclosing probabilistic hulls, i.e., a hull over all possible probability distributions when some parameters are uncertain and do not have a probability distribution. We denote the probability density function (PDF) of the random process $\mathbf{X}(t)$ defined by (36) for a specific trajectory $u(t) \in \mathcal{U}$ at time $t = r$ by $f_{\mathbf{X}}(x, r)$. The *enclosing probabilistic hull* (EPH) of all possible probability density functions $f_{\mathbf{X}}(x, r)$ is denoted by $\bar{f}_{\mathbf{X}}(x, r)$ and defined as: $\bar{f}_{\mathbf{X}}(x, r) = \sup\{f_{\mathbf{X}}(x, r) | \mathbf{X}(t)$ is a solution of (36) $\forall t \in [0, r]$, $u(t) \in \mathcal{U}$, $f_{\mathbf{X}}(x, 0) = f_0\}$. The enclosing probabilistic hull for a time interval is defined as $\bar{f}_{\mathbf{X}}(x, [0, r]) = \sup\{\bar{f}_{\mathbf{X}}(x, t) | t \in [0, r]\}$.

Let us demonstrate the class `linearSys` by an example:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -1 & -4 \\ 4 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} u + \begin{bmatrix} 0.7 & 0 \\ 0 & 0.7 \end{bmatrix} \xi$$

```
% system matrices
A = [-1 -4; 4 -1];
B = eye(2);
C = 0.7*eye(2);

% linear system
sys = linProbSys('twoDimSys',A,B,C);
```

#### 4.2.4.1 Operation `reach`

Reachability analysis for linear probabilistic systems is similar to reachability analysis of linear systems without stochastic uncertainty. The main difference is that the solution for time intervals has to be enclosed by the aforementioned *enclosing probabilistic hulls* [44].

The settings for reachability analysis are specified as fields of the struct `options` (see Sec. 4.1.1). For stochastic linear systems, the following settings are available:

- `.timeStep` — time step size.
- `.taylorTerms` — number of Taylor terms for the computation of the exponential matrix $e^{A\Delta t}$ (see [33, Sec. 4.2.4]).
- `.zonotopeOrder` — upper bound for the zonotope order $\rho$ (see Sec. 2.2.1.1).
- `.reductionTechnique` — string specifying the method used to reduce the zonotope order (see Tab. 4). The default value is `'girard'`.

- `.gamma` scalar value specifying the size of the confidence set of normal distributions. The probability outside the confidence set is not computed, but added as a global probability of entering an unsafe set as discussed in [33, Sec. 4.2.3].

### 4.2.5 Linear ARX Models

Discrete-time autoregressive models with exogenous Input (ARX models) are considered in the `linearARX` class. They can be described by the following equation:

$$y[i] = \sum_{j=1}^{p} \bar{A}_j y[i-j] + \sum_{j=0}^{p} \bar{B}_j u[i-j] \tag{37}$$

where $y[i] \in \mathbb{R}^o$ is the system output at time step $i$ and $u[i] \in \mathbb{R}^m$ is the system input which can also include disturbances and measurements noise. An object of class `linearARX` can be constructed as follows:

$$\texttt{sys} = \texttt{linearARX}(\texttt{A\_bar}, \texttt{B\_bar}, \Delta t)$$
$$\texttt{sys} = \texttt{linearARX}(\texttt{name}, \texttt{A\_bar}, \texttt{B\_bar}, \Delta t)$$

where `name` is a string specifying the name of the system, `A_bar` and `B_bar` are cell arrays which contain the matrices $\bar{A}_j$ and $\bar{B}_j$, respectively, and $\Delta t$ is the sampling time specifying the time difference between the time steps $k$ and $k+1$.

An example for the class `linearARX` is given in the following:

$$\begin{bmatrix} y_1[i] \\ y_2[i] \end{bmatrix} = \begin{bmatrix} 0.4 & 0.6 \\ 0.6 & 0.4 \end{bmatrix} \begin{bmatrix} y_1[i-1] \\ y_2[i-1] \end{bmatrix} + \begin{bmatrix} 0.1 & 0 \\ 0.2 & 0.1 \end{bmatrix} \begin{bmatrix} y_1[i-2] \\ y_2[i-2] \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} u[i] + \begin{bmatrix} 0.3 \\ -0.7 \end{bmatrix} u[i-1] + \begin{bmatrix} 0.1 \\ 0 \end{bmatrix} u[i-2]$$

```
% system matrices
A_bar = {[0.4 0.6; 0.6 0.4];
    [0.1 0; 0.2 0.1]};
B_bar = {[0; 0];[0.3; -0.7];
    [0.1; 0]};

% sampling time
dt = 0.1;

% ARX system
sys = linearARX(A_bar, B_bar, dt);
```

#### 4.2.5.1 Operation `reach`

The reachable set for a linear ARX model can be computed as described in [89].

As order reduction is not implemented yet, we only have to specify the reachability algorithm in the struct `options`:

- `.armaxAlg` reachability algorithm (`'exactAddition'` [89, Prop. 3], `'tvpEfficient'` [89, Thm. 3] or `'tvpGeneral'` [89, Thm. 2]). The default value is `'tvpGeneral'`.

### 4.2.6 Nonlinear Systems

Although a fairly large group of dynamic systems can be described by linear systems, the extension to nonlinear systems is an important step towards the analysis of more complex systems.

We consider general nonlinear continuous systems defined by the differential equation

$$\dot{x}(t) = f(x(t), u(t)), \tag{38}$$
$$y(t) = g(x(t), u(t)), \tag{39}$$

where $x(t) \in \mathbb{R}^n$ is the system state, $u(t) \in \mathbb{R}^m$ is the system input, $y(t) \in \mathbb{R}^o$ is the system output, and $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ and $g : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^o$ are sufficiently smooth.

Nonlinear systems are implemented by the class `nonlinearSys`. An object of class `nonlinearSys` can be constructed as follows:

$$\begin{aligned}
&\texttt{sys} = \texttt{nonlinearSys}(\texttt{fun}),\\
&\texttt{sys} = \texttt{nonlinearSys}(\texttt{name}, \texttt{fun}),\\
&\texttt{sys} = \texttt{nonlinearSys}(\texttt{fun}, n, m),\\
&\texttt{sys} = \texttt{nonlinearSys}(\texttt{name}, \texttt{fun}, n, m),\\
&\texttt{sys} = \texttt{nonlinearSys}(\texttt{fun}, \texttt{outFun}),\\
&\texttt{sys} = \texttt{nonlinearSys}(\texttt{name}, \texttt{fun}, \texttt{outFun}),\\
&\texttt{sys} = \texttt{nonlinearSys}(\texttt{fun}, n, m, \texttt{outFun}, o),\\
&\texttt{sys} = \texttt{nonlinearSys}(\texttt{name}, \texttt{fun}, n, m, \texttt{outFun}, o),
\end{aligned}$$

where `name` is a string specifying the name of the system, `fun` is a MATLAB function handle defining the function $f(x(t), u(t))$ in (38), $n$ is the number of states (see (38)), and $m$ is the number of inputs (see (38)), `outFun` is the output equation in (39) and $o$ is the number of outputs (see (39)). If the number of states $n$, the number of inputs $m$, and the number of outputs $o$ are not provided, they are automatically determined from the function handle `fun`. If no output equation is provided, we assume $y = x$. Let us demonstrate the class `nonlinearSys` by an example:

$$\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix} = \begin{bmatrix}x_2 + u\\ (1 - x_1^2)x_2 - x_1\end{bmatrix}$$

```
% differential equation f(x,u)
f = @(x,u) [x(2) + u;
            (1-x(1)^2)*x(2)-x(1)];

% nonlinear system
sys = nonlinearSys(f);
```

#### 4.2.6.1 Operation `reach`

Reachability analysis of nonlinear systems is much more complicated compared to linear systems, because many valuable properties are no longer valid. One of them is the superposition principle, which allows one to obtain the homogeneous and the inhomogeneous solution separately. Another advantage of linear systems is that the reachable set can be computed by a linear map in the absence of uncertain inputs. This makes it possible to exploit that geometric representations, such as ellipsoids, zonotopes, and polytopes, are closed under linear transformations, i.e., they are again mapped to ellipsoids, zonotopes, and polytopes, respectively. In CORA, reachability analysis of nonlinear systems is based on state-space abstraction. We consider abstraction by linear systems as presented in [33, Section 3.4] and by polynomial systems as presented in [38]. Since the abstraction causes additional errors, the abstraction errors are determined in an over-approximative way and added as an additional uncertain input to ensure an over-approximative computation.

A brief visualization of the overall concept for computing the reachable set is shown in Fig. 10. As in the previous approaches, the reachable set is computed iteratively for time intervals $t \in \tau_k = [k\,r, (k+1)r]$ where $k \in \mathbb{N}^+$. The procedure for computing the reachable sets of the consecutive time intervals is as follows:

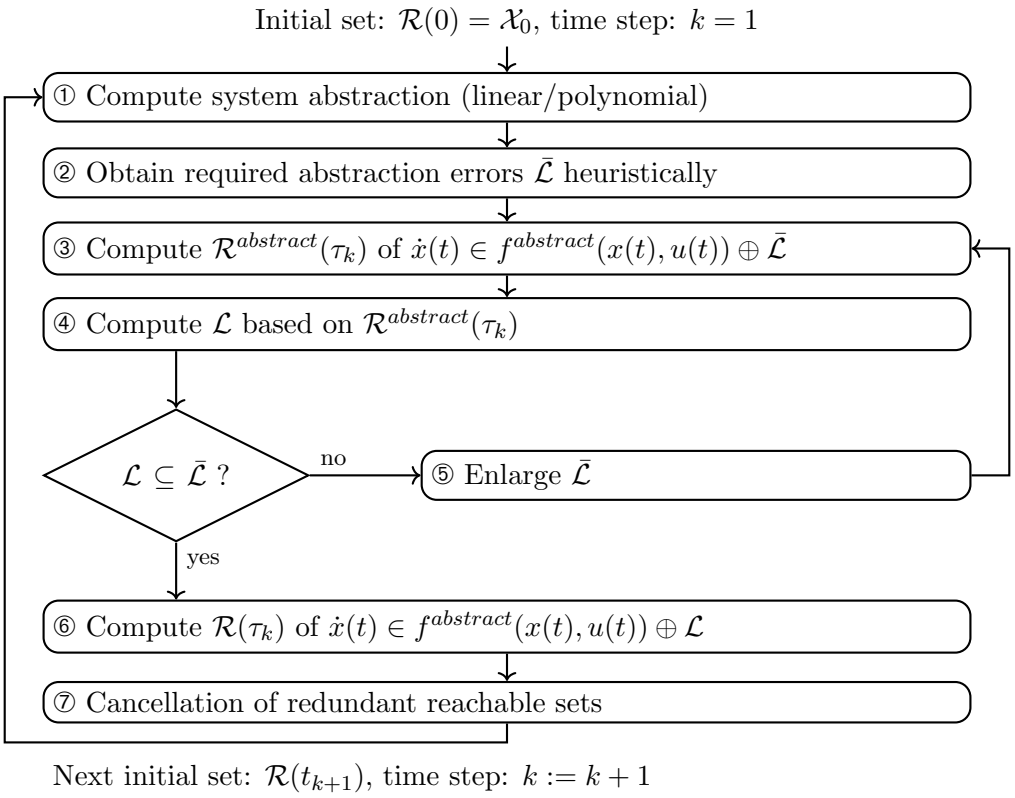



Figure 10: Computation of reachable sets for nonlinear systems – overview.

① The nonlinear system $\dot{x}(t) = f(x(t), u(t))$ is either abstracted to a linear system as shown in (33), or after introducing $z = [x^T, u^T]^T$, to a polynomial system resulting from the computation of a Taylor series of order $\kappa$:

$$\dot{x}_i \in \underbrace{\sum_{j=0}^{\kappa-1} \frac{\left((z(t) - z^*)^T \nabla\right)^j f_i(z^*)}{j!}}_{f_i^{abstract}(x,u)} \oplus \mathcal{L}_i(t), \tag{40}$$

where the Nabla operator is defined as $\nabla = \sum_{i=1}^{n+m} e_i \frac{\partial}{\partial z_i}$ with $e_i \in \mathbb{R}^{n+m}$ being orthogonal unit vectors. The set of abstraction errors $\mathcal{L}$ ensures that $f(x, u) \in f^{abstract}(x, u) \oplus \mathcal{L}$, which allows the reachable set to be computed in an over-approximative way.

② Next, the set of required abstraction errors $\bar{\mathcal{L}}$ is obtained heuristically.

③ The reachable set $\mathcal{R}^{abstract}(\tau_k)$ of $\dot{x}(t) \in f^{abstract}(x(t), u(t)) \oplus \bar{\mathcal{L}}$ is computed.

④ The set of abstraction errors $\mathcal{L}$ is computed based on the reachable set $\mathcal{R}^{abstract}(\tau_k)$.

⑤ As long as $\mathcal{L} \not\subseteq \bar{\mathcal{L}}$, the abstraction error is not admissible, requiring the assumption $\bar{\mathcal{L}}$ to be enlarged. If several enlargements are not successful, one has to split the reachable set and continue with one more partial reachable set.

⑥ If $\mathcal{L} \subseteq \bar{\mathcal{L}}$, the abstraction error is accepted and the reachable set is obtained by using the tighter abstraction error: $\dot{x}(t) \in f^{abstract}(x(t), u(t)) \oplus \mathcal{L}$.

⑦ It remains to increase the time step ($k := k + 1$) and cancel redundant reachable sets that are already covered by previously-computed reachable sets. This decreases the number of reachable sets that have to be considered in the next time interval.

The necessity of splitting reachable sets is indicated in the workspace outputs using the keyword `split`. In general, reachable sets of nonlinear systems are non-convex. Therefore, tight enclosures of the reachable set can often be better achieved by a non-convex set representation. For strongly nonlinear systems, we therefore recommend the conservative polynomialization algorithm (see Tab. 13) in combination with polynomial zonotopes (see Sec. 2.2.1.5).

Table 13: Reachability algorithms for nonlinear systems.

| **Algorithm** | **Description** | **Reference** |
|---|---|---|
| `lin` | conservative linearization | [15] and [33, Section 3.4] |
| `lin-adaptive` | conservative linearization with adaptive parameter tuning | [90] |
| `poly` | conservative polynomialization | [38] |
| `poly-adaptive` | conservative polynomialization with adaptive parameter tuning | [90] |
| `linRem` | abstraction by linear parametric system | – |

The settings for reachability analysis are specified as fields of the struct `options` (see Sec. 4.1.1). The following settings are available. (Note that all but `options.alg` are redundant if `options.alg` is set to `lin-adaptive` or `poly-adaptive`):

- `.alg` string specifying the used reachability algorithm (see Tab. 13).
- `.timeStep` time step size for one reachability time step.
- `.tensorOrder` order $\kappa$ of the Taylor series expansion for the abstraction in (40) of the dynamic function. The recommended values are $\kappa = 2$ or $\kappa = 3$.
- `.tensorOrderOutput` order $\kappa$ of the Taylor series expansion for the abstraction in (40) of the output function. The recommended values are $\kappa = 2$ or $\kappa = 3$.
- `.taylorTerms` number of Taylor terms for the computation of the exponential matrix $e^{A\Delta t}$ (see [33, Eq. (3.2)]) for the linearized system.
- `.zonotopeOrder` upper bound for the zonotope order $\rho$ (see Sec. 2.2.1.1).
- `.reductionTechnique` string specifying the method used to reduce the zonotope order (see Tab. 4). The default value is `'girard'`.
- `.errorOrder` the zonotope order $\rho$ (see Sec. 2.2.1.1) is reduced to `errorOrder` internally before the linearization error is computed. This is done since the computation of the linearization error involves quadratic or even cubic maps that drastically increase the number of generators of the set.
- `.intermediateOrder` upper bound for the zonotope order $\rho$ (see Sec. 2.2.1.1) during internal computations of the algorithm.
- `.maxError` vector of dimension $\mathbb{R}^n$ specifying the upper bound for the admissble abstraction error $\mathcal{L}$ for each system dimension. If the abstraction error exceeds the bound, the reachable set is splitted (see Step. 5 in Fig. 10). The default value is $\infty$ (no splitting).

- `.reductionInterval` number of time steps after which redundant sets resulting from splitting are cancelled (see Step. 7 in Fig. 10). The default value is $\infty$ (no cancellation).
- `.lagrangeRem` struct containing settings for evaluating the Lagrange remainder $\mathcal{L}$ (see Tab. 27).
- `.polyZono` struct containing settings for restructuring polynomial zonotopes (see Tab. 25). Only to be used for algorithm `'poly'` and if polynomial zonotopes are used to represent the reachable set.

#### 4.2.6.2 Operation `reachInner`

To compute inner-approximations of reachable sets for nonlinear systems, CORA implements three different algorithms: The set scaling approach from [91] (`options.algInner = 'scale'`) which represents inner-approximations with polynomial zonotopes, the approach from [92] (`options.algInner = 'parallelo'`) that represents inner-approximations with parallelotopes, and the Picard-Lindelöf iteration based approach from [93] (`options.algInner = 'proj'`). While the algorithms `'parallelo'` and `'scale'` compute full inner-approximations, the algorithm `'proj'` only computes an inner-approximation of the projection onto a single dimension.

The settings for reachability analysis are specified as fields of the struct `options` (see Sec. 4.1.2). The following settings are available:

- `.algInner` string specifying the used reachability algorithm. The available algorithms are `'scale'` (algorithm from [91]), `'parallelo'` (algorithm [92]), and `'proj'` (algorithm from [93]).

**Settings for Scaling Algorithm:**

- `.timeStep` time step size for one reachability time step for the computation of the outer-approximation (see Sec. 4.2.6.1).
- `.timeStepInner` time step size for the inner-approximation, which has to be a multiple of the time step size for the outer-approximation. If set to `'end'`, only the inner-approximation at the final time is computed. The default value is `'end'`.
- `.contractor` string specifying the contractor that is applyied to properly scale the reachable set (see Tab. 28). The default value is `'linearize'`.
- `.orderInner` zonotope order $\rho$ (see Sec. 2.2.1.1) for the inner-approximation. The default value is 5.
- `.splits` number of recusive splits for the contraction (see Sec. 7.8). The default value is 8.
- `.iter` number of consequtive contractions applied (see Sec. 7.8). The default value is 2.

- `.scaleFac` — scaling factor $\in ]0, 1]$ applied to scale the initial guess determined with nonlinear programming. If set to `'auto'` the optimal scaling factor is determined automatically. The default value is `'auto'`.
- `.inpChanges` — number of input changes of the piecewise-constant input signals used to approximate time-varying inputs. The default value is 0.
- `.taylorTerms` — setting for computing the outer-approximation of the reachable set (see Sec. 4.2.6.1).
- `.zonotopeOrder` — setting for computing the outer-approximation of the reachable set (see Sec. 4.2.6.1).
- `.reductionTechnique` — setting for computing the outer-approximation of the reachable set (see Sec. 4.2.6.1).
- `.errorOrder` — setting for computing the outer-approximation of the reachable set (see Sec. 4.2.6.1).
- `.intermediateOrder` — setting for computing the outer-approximation of the reachable set (see Sec. 4.2.6.1).
- `.polyZono` — setting for computing the outer-approximation of the reachable set (see Sec. 4.2.6.1).

**Settings for Parallelotope Algorithm:**

The settings are identical to the settings for reachability analysis of nonlinear systems, which are documented in Sec. 4.2.6.1.

**Settings for Projection Algorithm:**

- `.timeStep` — time step size for one reachability time step for the computation of the outer-approximation as well as the inner-approximation.
- `.taylorOrder` — Taylor order $k$ for the Taylor series expansion in solution space (see [93, Eq. (6)] and [93, Alg. 1]).
- `.taylmOrder` — upper bound for the polynomial degree of the Taylor model monomials (see Sec. 2.2.3.1)

### 4.2.7 Nonlinear Systems with Uncertain Parameters

Nonlinear parametric systems extend nonlinear systems by additionally considering uncertain parameters $p$:

$$\dot{x}(t) = f(x(t), u(t), p), \quad p \in \mathcal{P} \subset \mathbb{R}^p, \tag{41}$$

$$y(t) = g(x(t), u(t), p), \tag{42}$$

where $x(t) \in \mathbb{R}^n$ is the system state, $u(t) \in \mathbb{R}^m$ is the system input, $p \in \mathbb{R}^p$ is the parameter vector, $y(t) \in \mathbb{R}^o$ is the system output, and $f : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^p \to \mathbb{R}^n$ and $g : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^p \to \mathbb{R}^o$

are sufficiently smooth. As for linear parametric systems (see Sec. 4.2.2), the parameters $p \in \mathcal{P}$ can be constant over time or time-varying.

Nonlinear parametric systems are implemented by the class `nonlinParamSys`. An object of class `nonlinearSys` can be constructed as follows:

$$
\begin{aligned}
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{fun}),\\
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{fun}, \texttt{type}),\\
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{name}, \texttt{fun}),\\
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{name}, \texttt{fun}, \texttt{type}),\\
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{fun}, n, m, p),\\
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{fun}, n, m, p, \texttt{type}),\\
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{name}, \texttt{fun}, n, m, p),\\
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{name}, \texttt{fun}, n, m, p, \texttt{type}),\\
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{fun}, \texttt{outFun}),\\
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{fun}, \texttt{type}, \texttt{outFun}),\\
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{name}, \texttt{fun}, \texttt{outFun}),\\
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{name}, \texttt{fun}, \texttt{type}, \texttt{outFun}),\\
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{fun}, n, m, p, \texttt{outFun}, o),\\
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{fun}, n, m, p, \texttt{type}, \texttt{outFun}, o),\\
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{name}, \texttt{fun}, n, m, p, \texttt{outFun}, o),\\
&\texttt{sys} = \texttt{nonlinParamSys}(\texttt{name}, \texttt{fun}, n, m, p, \texttt{type}, \texttt{outFun}, o),
\end{aligned}
$$

where `name` is a string specifying the name of the system, `fun` is a MATLAB function handle defining the function $f(x(t), u(t), p)$ in (41), $n$ is the number of states (see (41)), $m$ is the number of inputs (see (41)), $p$ is the number of parameters (see (41)), `type` is a string that specifies if the parameter are constant over time (`'constParam'`) or time-varying (`'varParam'`), `outFun` is a MATLAB function handle defining the function $g(x(t), u(t), p)$ in (42), and $o$ is the number of outputs (see (42)). The default value for `type` is `'constParam'`. If the number of states $n$, the number of inputs $m$, the number of parameters $p$, and the number of outputs $o$ are not provided, they are automatically determined from the function handle `fun`. If no output equation is provided, we assume $y = x$. Let us demonstrate the class `nonlinParamSys` by an example:

$$
\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 + u \\ p(1 - x_1^2)x_2 - x_1 \end{bmatrix}
$$

```
% differential equation f(x,u,p)
f = @(x,u,p) [x(2) + u;
              p*(1-x(1)^2)*x(2)-x(1)];

% nonlinear parametric system
sys = nonlinParamSys(f);
```

An alternative to nonlinear parametric systems with constant parameters is to define each parameter as a state variable $\tilde{x}_i$ with the trivial dynamics $\dot{\tilde{x}}_i = 0$. Time-varying parameters can be equivalently modeled as uncertain inputs. For both cases the result is a nonlinear system that can be handled as described in Sec. 4.2.6. The question whether to compute the solution with the dedicated approach presented in this section or with the approach for nonlinear systems has not yet been thoroughly investigated.

#### 4.2.7.1 Operation `reach`

For reachability analysis of nonlinear parametric systems we use the same algorithms and settings as for nonlinear systems (see Sec. 4.2.6.1). The only difference is that the conservative

polynomialization algorithm [38] (`options.alg = 'poly'`) is yet only implemented for parametic systems for which the set of uncertain parameters $\mathcal{P}$ (see (41)) is a single point instead of a set.

### 4.2.8 Nonlinear Discrete-Time Systems

In this section, we consider nonlinear discrete-time systems defined as

$$x[i+1] = f\big(x[i], u[i]\big), \tag{43}$$
$$y[i] = g\big(x[i], u[i]\big), \tag{44}$$

where $x[i] \in \mathbb{R}^n$ is the system state, $u[i] \in \mathbb{R}^m$ is the system input, $y[i]$ is the system output, and $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ and $g : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^o$ are continuous functions. Nonlinear discrete-time systems are implemented in CORA by the class `nonlinearSysDT`. An object of class `nonlinearSysDT` can be constructed as follows:

$$\begin{aligned}
&\texttt{sys} = \texttt{nonlinearSysDT}(\texttt{fun}, \Delta t),\\
&\texttt{sys} = \texttt{nonlinearSysDT}(\texttt{name}, \texttt{fun}, \Delta t),\\
&\texttt{sys} = \texttt{nonlinearSysDT}(\texttt{fun}, \Delta t, n, m),\\
&\texttt{sys} = \texttt{nonlinearSysDT}(\texttt{name}, \texttt{fun}, \Delta t, n, m),\\
&\texttt{sys} = \texttt{nonlinearSysDT}(\texttt{fun}, \Delta t, \texttt{outFun}),\\
&\texttt{sys} = \texttt{nonlinearSysDT}(\texttt{name}, \texttt{fun}, \Delta t, \texttt{outFun}),\\
&\texttt{sys} = \texttt{nonlinearSysDT}(\texttt{fun}, \Delta t, n, m, \texttt{outFun}, o),\\
&\texttt{sys} = \texttt{nonlinearSysDT}(\texttt{name}, \texttt{fun}, \Delta t, n, m, \texttt{outFun}, o),
\end{aligned}$$

where `name` is a string specifying the name of the system, `fun` is a MATLAB function handle defining the function $f(x[i], u[i])$ in (43), $\Delta t$ is the sampling time specifying the time difference between $x[i+1]$ and $x[i]$, $n$ is the number of states (see (43)), and $m$ is the number of inputs (see (43)), `outFun` is a MATLAB function handle defining the function $g(x[i], u[i])$ in (44), and $o$ is the number of outputs. If the number of states $n$, the number of inputs $m$, and the number of outputs $o$ are not provided, they are automatically determined from the function handle `fun`. If no output equation is provided, we assume $y = x$. Let us demonstrate the class `nonlinearSysDT` by an example:

$$\begin{bmatrix} x_1[i+1] \\ x_2[i+1] \\ x_3[i+1] \end{bmatrix} = \begin{bmatrix} x_1[i] + u_1[i] \\ x_2[i] + u_2[i]\cos(x_1[i]) \\ x_3[i] + u_2[i]\sin(x_1[i]) \end{bmatrix}$$

```
% equation f(x,u)
f = @(x,u) [x(1) + u(1); ...
            x(2) + u(2)*cos(x(1)); ...
            x(3) + u(2)*sin(x(1))];

% sampling time
dt = 0.25;

% nonlinear discrete-time system
sys = nonlinearSysDT(f,dt);
```

#### 4.2.8.1 Operations `reach` / `observe`

Since the system evolves in discrete time, the task of calculating the reachable set is identical to the computation of the image of the nonlinear function $f(x[i], u[i])$ in (43) for $x[i] \in \mathcal{X}_i$ and $u[i] \in \mathcal{U}$. Similar to continuous-time nonlinear systems, we abstract the nonlinear function by a

Taylor series of order $\kappa$:

$$x_l[i+1] \in \underbrace{\sum_{j=0}^{\kappa-1} \frac{\left((z[i]-z^*)^T\nabla\right)^j f_l(z^*)}{j!}}_{f_l^{abstract}(x[i],u[i])} \oplus \mathcal{L}_l[i], \tag{45}$$

where $z[i] = [x[i]^T \; u[i]^T]^T$ and the Nabla operator is defined as $\nabla = \sum_{i=1}^{n+m} e_i \frac{\partial}{\partial z_i}$ with $e_i \in \mathbb{R}^{n+m}$ being orthogonal unit vectors. The set of abstraction errors $\mathcal{L}$ ensures that $f(x,u) \in f^{abstract}(x[i],u[i]) \oplus \mathcal{L}$, which allows the reachable set to be computed in an over-approximative way.

The settings for reachability analysis are specified as fields of the struct `options` (see Sec. 4.1.1). For nonlinear discrete-time systems the following settings are available:

- `.tensorOrder` — order $\kappa$ of the Taylor series expansion for the abstraction of the dynamic function in (45). The recommended values are $\kappa = 2$ or $\kappa = 3$.
- `.tensorOrderOutput` — order $\kappa$ of the Taylor series expansion for the abstraction of the output function. The recommended values are $\kappa = 2$ or $\kappa = 3$.
- `.zonotopeOrder` — upper bound for the zonotope order $\rho$ (see Sec. 2.2.1.1).
- `.reductionTechnique` — string specifying the method used to reduce the zonotope order (see Tab. 4). The default value is `'girard'`.
- `.errorOrder` — the zonotope order $\rho$ (see Sec. 2.2.1.1) is reduced to `errorOrder` internally before the linearization error is computed. This is done since the computation of the linearization error involves quadratic or even cubic maps that drastically increase the number of generators of the set.
- `.lagrangeRem` — struct containing settings for evaluating the Lagrange remainder $\mathcal{L}$ (see Tab. 27).
- `.polyZono` — struct containing settings for restructuring polynomial zonotopes (see Tab. 25). Only to be used in `reach` for algorithm `'poly'` and if polynomial zonotopes are used to represent the reachable set.

Besides the above settings for reachability analysis, one can also specify the algorithm for set-based estimation. The current list of observers for discrete-time nonlinear systems implemented is shown in Tab. 18. The implemented observers are categorized according to [54, 55]. While some reachability analysis approaches are agnostic with respect to the set representation, most approaches for set-based observers are specifically designed for a specific set representation.

Because strip-based observers can only finish their computation of the estimated set after the measurement, their result is always delayed. When a set-propagation observer or interval observer is real-time capable, the estimated set is obtained ahead of time. This issue can be fixed for strip-based observers when additionally computing a one-step prediction and use this set as the initial set as shown in [71, Sec. III]. For this reason, we list these algorithms as not in Tab. 10.

#### 4.2.8.2 Operation `isconform`

The operator `isconform` checks reachset conformance. The settings for reachset conformance checking are as for reachability analysis and rapidly-exploring random trees. In addition, the

Table 18: Algorithms for set-based estimation for discrete-time nonlinear systems.

| Technique | Set representation | Ready for control | Supported Reference |
|---|---|---|---|
| strip-based observers | | | |
| VolMin-A | zonotope | ✗ | [72] |
| VolMin-B | zonotope | ✗ | [73] |
| FRad-A | zonotope | ✗ | [72] |
| FRad-B | zonotope | ✗ | [74] |
| CZN-A | constr. zono. | ✗ | [34] |
| CZN-B | constr. zono. | ✗ | [79] |
| set-propagation observers | | | |
| FRad-C | zonotope | ✓ | [82] |

algorithm as listed in Tab. 19 can be selected.

Table 19: Conformance algorithms for discrete-time nonlinear systems.

| Algorithm | Description | Reference |
|---|---|---|
| `RRT` | checks conformance using RRTs | [60] |
| `BF` | brute-force conformance check | [53] |

### 4.2.9 Nonlinear ARX Models

In this section, we consider nonlinear autoregressive models with exogenous input (nonlinear ARX models) defined as

$$y[i] = f(y[i-1:i-p], u[i:i-p]) \tag{46}$$

with

$$y[i-1:i-p] = \begin{bmatrix} y[i-1]^\top & \dots & y[i-p]^\top \end{bmatrix}^\top,$$
$$u[i:i-p] = \begin{bmatrix} u[i]^\top & \dots & u[i-p]^\top \end{bmatrix}^\top,$$

where $y[i] \in \mathbb{R}^o$ is the system output, $u[i] \in \mathbb{R}^m$ is the system input, and $f : \mathbb{R}^{op} \times \times \mathbb{R}^{m(p+1)} \to \mathbb{R}^o$ is a continuous function. Nonlinear ARX models are implemented in CORA by the class `nonlinearARX`. An object of class `nonlinearARX` can be constructed as follows:

$$\texttt{sys} = \texttt{nonlinearARX}(\texttt{fun}, \Delta t, o, m, p),$$
$$\texttt{sys} = \texttt{nonlinearARX}(\texttt{name}, \texttt{fun}, \Delta t, o, m, p),$$

where `name` is a string specifying the name of the system, `fun` is a MATLAB function handle defining the function $f(y[i-1:i-p], u[i:i-p])$ in (46), $\Delta t$ is the sampling time specifying the time difference between $y[i]$ and $y[i-1]$, $o$ is the dimension of the system output $y[i]$, $m$ is the dimension of the system input $u[i]$, and $p$ is the number of past outputs and inputs that are considered in the function $f$. Let us demonstrate the class `nonlinearARX` by an example:

$$\begin{bmatrix} y_1[i] \\ y_2[i] \\ y_3[i] \end{bmatrix} = \begin{bmatrix} y_1[i-1] + u_1[i] + \cos(u_1[i-1]) \\ y_2[i-2] + y_3[i-1]\cos(u_2[i]) \\ y_3[i-1] + u_2[i-1]\sin(y_1[i-2]) \end{bmatrix}$$

```
% equation f(y,u)
f = @(y,u) [y(1) + u(1)*sin(u(3)); ...
            y(5) + y(3)*cos(u(2)); ...
            y(3) + u(4)*sin(y(4))];

% sampling time
dt = 0.25;

% nonlinear discrete-time system
sys = nonlinearARX(f,dt,3,2,2);
```

#### 4.2.9.1 Operations `reach`

By introducing the state $x[i] = y[i : i-p+1]$ and the input $\tilde{u}[i] = u[i : i-p]$, the nonlinear ARX model in (46) can be transformed to an object of the class `nonlinearSysDT`. More details on the transformation can be found in [6]. Thus, the reachable set of `nonlinearARX` objects can be computed analogously to the reachable set of `nonlinearSysDT` objects leading to the same available settings specified in the struct `options` (see Sec. 4.2.8).

### 4.2.10 Nonlinear Differential-Algebraic Systems

The class `nonlinDASys` considers time-invariant, semi-explicit, index-1 differential-algebraic systems defined as

$$\begin{aligned} \dot{x} &= f(x(t), y(t), u(t)), \\ 0 &= g(x(t), y(t), u(t)), \\ z &= h(x(t), y(t), u(t)), \end{aligned} \tag{47}$$

where $x(t) \in \mathbb{R}^n$ is the vector of differential variables, $y(t) \in \mathbb{R}^q$ is the vector of algebraic variables, $u(t) \in \mathbb{R}^m$ is the vector of inputs, $z(t) \in \mathbb{R}^o$ is the system output, and $f : \mathbb{R}^n \times \mathbb{R}^q \times \mathbb{R}^m \to \mathbb{R}^n$, $g : \mathbb{R}^n \times \mathbb{R}^q \times \mathbb{R}^m \to \mathbb{R}^q$, and $h : \mathbb{R}^n \times \mathbb{R}^q \times \mathbb{R}^m \to \mathbb{R}^o$ are sufficiently smooth continuous functions. The initial state is consistent when $g(x(0), y(0), u(0)) = 0$, while for DAEs with an index greater than 1, further hidden algebraic constraints have to be considered [94, Chapter 9.1]. For an implicit DAE, the index-1 property holds if and only if $\forall t : \det(\frac{\partial g(x(t),y(t),u(t))}{\partial y}) \neq 0$, i.e., the Jacobian of the algebraic equations is non-singular [95, p. 34]. Loosely speaking, the index specifies the distance to an ODE (which has index 0) by the number of required time differentiations of the general form $0 = F(\dot{\tilde{x}}, \tilde{x}, u, t)$ along a solution $\tilde{x}(t)$, in order to express $\dot{\tilde{x}}$ as a continuous function of $\tilde{x}$ and $t$ [94, Chapter 9.1].

Nonlinear differential-algebraic systems are implemented by the class `nonlinDASys`. An object of class `nonlinDASys` can be constructed as follows:

$$\begin{aligned} &\texttt{sys} = \texttt{nonlinDASys}(\texttt{dynFun}, \texttt{conFun}), \\ &\texttt{sys} = \texttt{nonlinDASys}(\texttt{name}, \texttt{dynFun}, \texttt{conFun}), \\ &\texttt{sys} = \texttt{nonlinDASys}(\texttt{dynFun}, \texttt{conFun}, n, m, q), \\ &\texttt{sys} = \texttt{nonlinDASys}(\texttt{name}, \texttt{dynFun}, \texttt{conFun}, n, m, q), \\ &\texttt{sys} = \texttt{nonlinDASys}(\texttt{dynFun}, \texttt{conFun}, \texttt{outFun}), \\ &\texttt{sys} = \texttt{nonlinDASys}(\texttt{name}, \texttt{dynFun}, \texttt{conFun}, \texttt{outFun}), \\ &\texttt{sys} = \texttt{nonlinDASys}(\texttt{dynFun}, \texttt{conFun}, n, m, q, \texttt{outFun}, o), \\ &\texttt{sys} = \texttt{nonlinDASys}(\texttt{name}, \texttt{dynFun}, \texttt{conFun}, n, m, q, \texttt{outFun}, o), \end{aligned}$$

where `name` is a string specifying the name of the system, `dynFun` is a MATLAB function handle defining the function $f(x(t), y(t), u(t))$ in (47), `conFun` is a MATLAB function handle defining the function $g(x(t), y(t), u(t))$ in (47), `outFun` is a MATLAB function handle defining

the function $h(x(t), y(t), u(t))$ in (47), $n$ is the number of states (see (47)), $m$ is the number of inputs (see (47)), $q$ is the number of algebraic constraints (see (47)), and $o$ is the number of outputs (see (47). If the number of states $n$, the number of inputs $m$, the number of constraints $q$, and the number of outputs $o$ are not provided, they are automatically determined from the function handles `dynFun` and `conFun`. If no output equation is provided, we assume $z = x$. Let us demonstrate the class `nonlinDASys` by an example:

$$\dot{x} = x + 1 + u$$
$$0 = (x+1)y + 2$$

```
% differential equation f(x,y,u)
f = @(x,y,u) x + 1 + u;

% constraint equation g(x,y,u)
g = @(x,y,u) (x+1)*y + 2;

% nonlinear differential-algebraic system
sys = nonlinDASys(f,g);
```

Parametric uncertainties as demonstrated in Sec. 4.2.7 have not yet been implemented, but one can consider uncertain parameters using the existing techniques: for uncertain but fixed parameters, one can define each parameter as a state variable $\tilde{x}_i$ with the trivial dynamics $\dot{\tilde{x}}_i = 0$ and for time-varying parameters, one can specify the parameter as an uncertain input.

#### 4.2.10.1 Operation `reach`

For nonlinear differential-algebraic systems, CORA uses the algorithm in [96] to compute the reachable set. To apply the methods presented in Sec. 4.2.6.1, the algorithm performs an abstraction of the original nonlinear DAEs to linear differential inclusions for each consecutive time interval $\tau_k$. A different abstraction is used for each time interval to minimize the over-approximation error. Based on a linearization of the functions $f(x(t), y(t), u(t))$ and $g(x(t), y(t), u(t))$, one can abstract the dynamics of the original nonlinear DAE by a linear system plus additive uncertainty as detailed in [96, Section IV]. This linear system only contains dynamic state variables $x$ and uncertain inputs $u$. The algebraic state $y$ is obtained afterwards by the linearized constraint function $g(x(t), y(t), u(t))$ as described in [96, Proposition 2].

In contrast to ordinary differential equations, the initial state for differential-algebraic systems is not automatically consistent. One therefore has to specify a guess for a consistent initial algebraic state with the additional parameter `params.y0guess` (see Sec. 4.1). Depending on the guess, a consistent initial algebraic state is found using the Newton-Raphson method.

The settings for reachability analysis are specified as fields of the struct `options` (see Sec. 4.1.1). For nonlinear differential-algebraic systems, the following settings are available:

- `.timeStep` — time step size for one reachability time step.
- `.tensorOrder` — order $\kappa$ of the Taylor series expansion for the abstraction in [96, eq. (8)]. The recommended values are $\kappa = 2$ or $\kappa = 3$.
- `.tensorOrderOutput` — order $\kappa$ of the Taylor series expansion for the abstraction of the output function. The recommended values are $\kappa = 2$ or $\kappa = 3$.
- `.taylorTerms` — number of Taylor terms for the computation of the exponential matrix $e^{A\Delta t}$ (see [33, Eq. (3.2)]) for the linearized system.
- `.zonotopeOrder` — upper bound for the zonotope order $\rho$ (see Sec. 2.2.1.1).
- `.reductionTechnique` — string specifying the method used to reduce the zonotope order (see Tab. 4). The default value is `'girard'`.

- `.errorOrder` the zonotope order $\rho$ (see Sec. 2.2.1.1) is reduced to `errorOrder` internally before the linearization error is computed. This is done since the computation of the linearization error involves quadratic or even cubic maps that drastically increase the number of generators of the set.
- `.intermediateOrder` upper bound for the zonotope order $\rho$ (see Sec. 2.2.1.1) in internal computations of the algorithm.
- `.maxError_x` vector of dimension $\mathbb{R}^n$ specifying the upper bound for the admissible abstraction error $\mathcal{L}$ for each system dimension of the dynamic equation $f(x(t), y(t), u(t))$ (see (47)). If the abstraction error exceeds the bound, the reachable set is split (see Step. 5 in Fig. 10). The default value is $\infty$ (no splitting).
- `.maxError_y` vector of dimension $\mathbb{R}^q$ specifying the upper bound for the admissible abstraction error $\mathcal{L}$ for each system dimension of the constraint equation $g(x(t), y(t), u(t))$ (see (47)). If the abstraction error exceeds the bound, the reachable set is split (see Step. 5 in Fig. 10). The default value is $\infty$ (no splitting).
- `.reductionInterval` number of time steps after which redundant sets resulting from splitting are cancelled (see Step. 7 in Fig. 10). The default value is $\infty$ (no cancellation).
- `.lagrangeRem` struct containing settings for evaluating the Lagrange remainder $\mathcal{L}$ (see Tab. 27).

### 4.2.11 Neural Network Control Systems

Due to the current trend towards artificial intelligence, the system class of neural network control system is gaining more and more importance. In CORA, we consider neural network control systems with discrete feedback, where the neural network controller updates the control input at the end of the time steps defined by the sampling time $\Delta t$. CORA implements the approach described in [97, 98], please visit Sec. 6 for implementation details on the neural network.

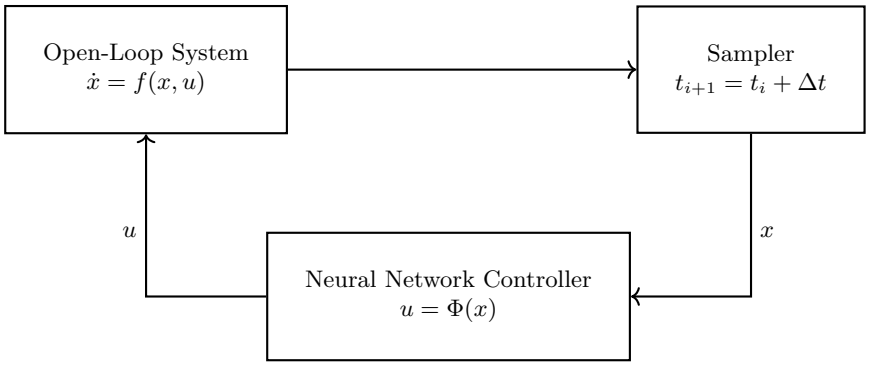



Figure 11: Structure of a neural network control system.

Neural network control systems are implemented by the class `neurNetContrSys`. An object of

class `neurNetContrSys` can be constructed as follows:

$$\texttt{sys = neurNetContrSys(sysOL,nn,}\Delta t\texttt{)},$$

where `sysOL` is a child of class `contDynamics` (see Fig. 1) describing the dynamics of the open-loop system, `nn` is the neural network controller of class `neuralNetwork`, and $\Delta t \in \mathbb{R}_{>0}$ is the sampling time of the controller. Note that the dynamics of the open-loop system can be specified by any of the system classes described in Sec. 4.2 (e.g. `linearSys`, `nonlinearSys`, etc.). Since CORA does not explicitly distinguish between control inputs and uncertain inputs, we treat the first $N$ inputs of the open-loop system as control inputs and the remaining inputs as uncertain inputs, where $N$ is the number of outputs of the neural network controller. Let us demonstrate the class `neurNetContrSys` by an example:

```
% open-loop system
f = @(x,u) [x(2) + u(2); (1-x(1)^2)*x(2) - x(1) + u(1)];
sysOL = nonlinearSys(f);

% neural network controller
W1 = rand(100,2); b1 = rand(100,1);
W2 = rand(1,100); b2 = rand(1,1);
nn = neuralNetwork({
    nnLinearLayer(W1, b1);
    nnReLULayer();
    nnLinearLayer(W2, b2);
    nnReLULayer();
})

% neural network controlled system
dt = 0.01;
sys = neurNetContrSys(sysOL,nn,dt);
R = reach(sys, params, options);
```

where `params` and `options` are the settings for reachability analysis for `sysOL`, which are identical to the settings for the respective open-loop system class (Sec. 4.2). Note that the sampling time $\Delta t$ of the neural network can be different from the time step for reachability analysis stored in `options.timeStep`. Options for the network evaluations are stored in `options.nn` (see Sec. 6 for details). Further examples of neural network control systems can be found at `./cora/examples/contDynamics/neurNetContrSys`.

## 4.3 Hybrid Dynamics

Hybrid systems consist of a finite number of state space regions called *locations*, each of which with a specific continuous dynamics. Besides a continuous state $x$, there also exists a discrete state $v$ representing the current location. The continuous initial state may take values within continuous sets while only a single initial discrete state is assumed without loss of generality[18]. The switching of the continuous dynamics is triggered by *guard sets*. Jumps in the continuous state are considered after the discrete state has changed. One of the most intuitive examples where jumps in the continuous state can occur, is the bouncing ball example (see Fig. 12), where the velocity of the ball changes instantaneously when the ball hits the ground.

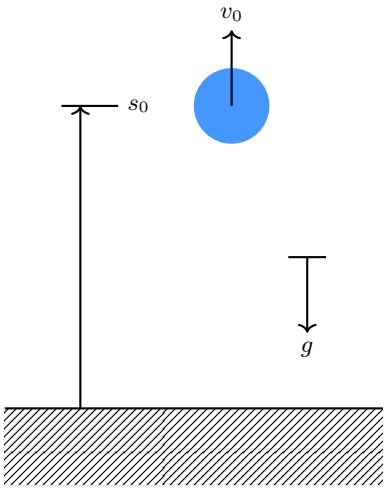
In CORA, hybrid systems are modeled by hybrid automata $HA = (L_1, \dots, L_p)$, which are defined by a finite list of locations $(L_1, \dots, L_p)$, where each location $L_i = (f_i(\cdot), \mathcal{S}_i, \mathbf{T}_i)$, $i = 1, \dots, p$ consists of

- a differential equation $\dot{x}(t) = f_i(\cdot)$ describing the continuous dynamics,

[18] In the case of several initial discrete states, the reachability analysis can be performed for each discrete state separately.

- an invariant set $\mathcal{S}_i \subset \mathbb{R}^n$ describing the region where the differential equation is valid,
- a list $\mathbf{T}_i = (T_1, \dots, T_q)$ of transitions $T_j = (\mathcal{G}_j, r_j(\cdot), d_j)$, $j = \{1, \dots, q\}$ from the current location to other locations, where $\mathcal{G}_j \subset \mathbb{R}^n$ is a guard set, $r_j \colon \mathbb{R}^n \to \mathbb{R}^n$ is a reset function, and $d_j \in \{1, \dots, p\}$ is the index of the target location.

The evolution of the hybrid automaton is described informally as follows: Starting from an initial location $v(0) \in \{1, \dots, p\}$ and an initial state $x(0) \in \mathcal{S}_{v(0)}$, the continuous state evolves according to the flow function $\dot{x}(t) = f_{v(0)}(\cdot)$ that is assigned to the location $v(0)$. If the continuous state is within a guard set $\mathcal{G}_j$ of a transition $T_j$, the transition $T_j$ can be taken and has to be taken if the state would otherwise leave the invariant $\mathcal{S}_{v(0)}$. When the transition from the previous location $v(0)$ to the next location $d_j$ is taken, the system state is updated according to the reset function $r_j(\cdot)$. Afterwards, the continuous state evolves according to the flow function of the next location.




$$
\begin{aligned}
HA &= (L_1) \\
L_1 &= (f_1(\cdot), \mathcal{S}_1, (T_1)) \\
f_1(x,u) &= \begin{bmatrix} x_2 \\ -g \end{bmatrix}, \quad g = 9.81 \\
\mathcal{S}_1 &= \left\{ [x_1 \ x_2]^T \in \mathbb{R}^2 \;\middle|\; x_2 \geq 0 \right\} \\
T_1 &= (\mathcal{G}_1, r_1(\cdot), 1) \\
\mathcal{G}_1 &= \left\{ [x_1 \ x_2]^T \in \mathbb{R}^2 \;\middle|\; x_1 = 0, \ x_2 \leq 0 \right\} \\
r(x) &= \begin{bmatrix} x_1 \\ -\alpha x_2 \end{bmatrix}, \quad \alpha = 0.75
\end{aligned}
$$

Figure 12: Example for a hybrid system: bouncing ball.

A simple example for a hybrid system is the bouncing ball shown in Fig. 12, where the continuous system states are the vertical position $x_1 = s$ and the vertical velocity $x_2 = v$, and $\alpha \in [0, 1]$ is the rebound factor that indirectly models the loss of energy during the collision with the ground. We will use the bouncing ball as a running example throughout this section.

Transitions between two locations are modeled in CORA by the class `transition`. An object of class transition can be constructed as follows:

$$
\begin{aligned}
T &= \texttt{transition}(\mathcal{G}, \texttt{reset}, d), \\
T &= \texttt{transition}(\mathcal{G}, \texttt{reset}, d, \texttt{label}),
\end{aligned}
\tag{48}
$$

where

- $\mathcal{G} \subset \mathbb{R}^n$ is the guard set. Guard sets can be modeled by all set representations described in Sec. 2.2. Most commonly, guard sets are modeled as `polytope` or `levelSet` objects. The guard set can also be left empty which results in an instantaneous transition, i.e., the guard set is active as soon as the location containing the transition of that guard set is entered. This feature is only advisable to be used in combination with synchronization labels (see below).
- `reset` is the reset function $r \colon \mathbb{R}^n \to \mathbb{R}^n$. CORA supports both linear and nonlinear reset functions, defined as

$$
r(x,u) = Ax + Bu + c, \qquad A \in \mathbb{R}^{n \times n}, B \in \mathbb{R}^{n \times m}, c \in \mathbb{R}^n, \tag{49}
$$

$$
r(x,u) = h(x,u), \qquad h \colon \mathbb{R}^n \to \mathbb{R}^n. \tag{50}
$$

The reset function is specified as a `linearReset` or `nonlinearReset` object. For linear reset functions, one initializes a `linearReset(A,B,c)` object with the matrices $A$, $B$, and the vector $c$ in (49). For nonlinear reset functions, the `nonlinearReset(f)` object is initialized with `f` storing a MATLAB function handle that defines the nonlinear reset function $h(x, u)$.

- $d \in \{1, \ldots, p\}$ is the index of the target location.
- `label` is the synchronization label (only class `parallelHybridAutomata`): All transitions with the same synchronization label are executed simulaneously under the condition that the corresponding guard sets of all transitions are triggered. Currently, CORA only allows one transition of the set of transitions with the same synchronization label to have a non-empty guard set. Consequently, all transitions trigger if the one guard set is triggered.

For the bouncing ball example in Fig. 12, the transition $T_1$ can be constructed as follows:

```
% guard set
guard = polytope([0 1],0,[1 0],0)

% reset function
reset = linearReset([1 0; 0 -0.75], [0; 0], [0;0]);

% transtition object
trans = transition(guard,reset,1);
```

The locations of a hybrid automaton are modeled in CORA by the class `location`. An object of class `location` can be constructed as follows:

$$\begin{aligned} L &= \texttt{location}(\mathcal{S}, \mathbf{T}, f(\cdot)), \\ L &= \texttt{location}(\texttt{name}, \mathcal{S}, \mathbf{T}, f(\cdot)), \end{aligned} \tag{51}$$

where

- `name` is a string that specifies the name of the location.
- $\mathcal{S} \subset \mathbb{R}^n$ is the invariant set. Invariant sets can be modeled by all set representations described in Sec. 2.2. Most commonly, guard sets are modeled as `polytope` or `levelSet` objects.
- $\mathbf{T} = (T_1, \ldots, T_j)$ is the list of transitions from the current location to other locations represented as an array of `transition` objects (see (48)).
- $\dot{x} = f(\cdot)$ is the differential equation that describes the continuous dynamics in the current location. The continous dynamics can be modeled by any of the system classes described in Sec. 4.2.

For the bouncing ball example in Fig. 12, the location $L_1$ can be constructed as follows:

```
% differential equation
sys = linearSys([0 1;0 0],[0;0],[0;-9.81]);

% invariant set
inv = polytope([-1 0],0);

% location object
loc = location(inv,trans,sys);
```

### 4.3.1 Hybrid Automata

A hybrid automaton is modeled by the class `hybridAutomaton`. An object of class `hybridAutomaton` can be constructed as follows:

$$\texttt{HA} = \texttt{hybridAutomaton}(\mathbf{L}),$$

where $\mathbf{L} = (L_1, \ldots, L_p)$ is a list of location objects represented as an array of `location` objects (see (51)).

The hybrid automaton for the bouncing ball example in Fig. 12 can be constructed as follows:

```
% list of locations
locs(1) = loc;

% hybrid automaton object
HA = hybridAutomaton(locs);
```

#### 4.3.1.1 Operation `reach`

For reachability analysis, we consider a set of initial states $\mathcal{X}_0 \subseteq \mathcal{S}_{v(0)}$ and a set of uncertain inputs $\mathcal{U} \subset \mathbb{R}^m$. The set of uncertain inputs can be different for each location of the hybrid automaton. An illustration of a reachable set of a hybrid automaton is provided in Fig. 13. To calculate the reachable set inside a single location, CORA uses the reachability algorithms for continuous systems described in Sec. 4.2. The most challenging part in reachability analysis for hybrid automata is the computation of the intersection between the reachable set and the guard set. CORA supports multiple methods for the calculation of guard intersections, which are listed in Tab. 22. For the intersection methods `polytope`, `zonoGirard`, `conZonotope`, and `nondetGuard` (see Tab. 22), the intersection with the guard set is enclosed by one or multiple oriented hyperrectangles. CORA supports the three methods listed in Tab. 21 to calculate the orientation of these hyperrectangles. The resulting hyperrectangles for the different enclosure methods are visualized in Fig. 14. If multiple enclosure methods are specified, the reachable set is enclosed by the intersection of all computed hyperrectangles (see Fig. 14 (right)).

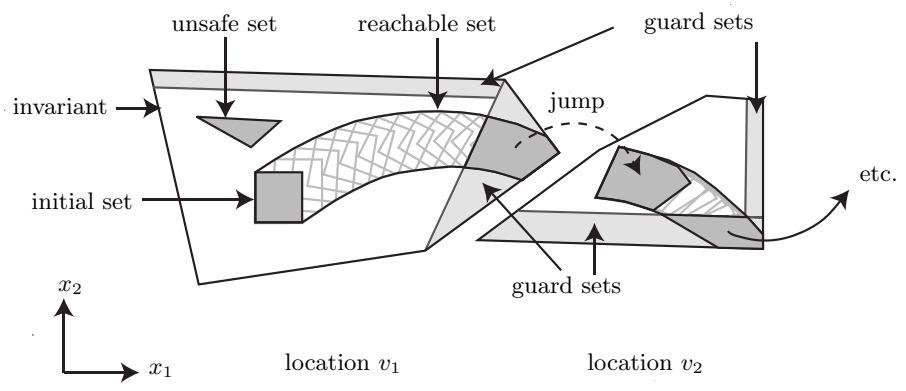



Figure 13: Illustration of the reachable set of a hybrid automaton.

The settings for reachability analysis are specified as fields of the struct `options` (see Sec. 4.1.1). For hybrid automata the settings for the involved continuous dynamics objects (see Sec. 4.2) have to be provided. In addition, the following settings specific to hybrid automata are available:

- `.guardIntersect` — string specifying the method used to calculate the intersections with the guard sets. The available methods are listed in Tab. 22.
- `.enclose` — cell array storing the strings that describe the methods for enclosing the intersections with the guard sets. The available methods are listed in Tab. 21. Required for the guard intersection methods `polytope`, `zonoGirard`, `conZonotope`, and `nondetGuard`.
- `.guardOrder` — upper bound for the zonotope order $\rho$ (see Sec. 2.2.1.1). The zonotope order is reduced to `guardOrder` before the intersections with the guard sets are calculated in order to decrease the computation time. Required for the guard intersection methods `conZonotope` and `hyperplaneMap`.
- `.timeStep` — time step size for one reachability time step. One can choose different time steps for each location by specifying `timeStep` as a cell array.
- `.intersectInvariant` — flag with value `true` or `false` specifying whether the computed reachable set is intersected with the invariant set to obtain a tighter enclosure. The default value is `false` (no intersection).

Furthermore, it is possible for hybrid automata to specify the set of uncertain inputs `params.U`, the time step `options.timeStep`, and the specification `spec` (see Sec. 4.1.1) as a MATLAB cell array with as many entries as the hybrid automaton has locations if the values are different for each location.

Table 21: Methods for enclosing guard intersections.

| Method | Description | Reference |
|---|---|---|
| `box` | The intersection is enclosed with an axis-aligned box. | Sec. V.A.a in [42] |
| `pca` | The orientation of the hyperrectangle is determined using principal component analysis. | Sec. V.A.b in [42] |
| `flow` | The orientation of the hyperrectangle is determined based on the direction of the flow of the dynamic function. | Sec. V.A.d in [42] |

### 4.3.2 Parallel Hybrid Automata

Complex systems can often be modeled as a connection of multiple distinct subcomponents, where each of these subcomponents represents a hybrid automaton. A naive approach to analyze these type of systems would be to construct a flat hybrid automaton from the interconnection of subcomponents (parallel composition, see e.g., [104, Def. 2.9]). This technique, however, requires calculating all possible combinations of subsystem locations, and therefore suffers from the curse of dimensionality: Consider for example a system consisting of 15 subcomponents, where each subcomponent has 10 discrete locations. The flat hybrid automaton for this system would consist of $10^{15}$ discrete locations.

This exponential increase in the number of locations can be avoided if the overall system is modeled as a parallel hybrid automaton. In this case, the system is described by a list of `hybridAutomaton` objects representing the subcomponents and by connections between these components. The flow function, the invariant set, and the guard sets for a location of the composed system are computed on-demand as soon as a simulated solution or the reachable set

Table 22: Guard intersection methods in CORA.

| Method | Description | Reference |
|---|---|---|
| `polytope` | The reachable sets are converted to polytopes and then intersected with the guard sets. Afterwards, the vertices of the sets representing the intersections are calculated. Finally, the vertices are enclosed by oriented hyperrectangles, where the orientation is determined by the methods in Tab. 21. | [99] |
| `zonoGirard` | First, suitable template directions are determined using the methods in Tab. 21. Then, the algorithm described in [100] is applied to compute an upper and a lower bound for the projection of the intersection between reachable set and guard set onto each template direction. | [100] |
| `conZonotope` | Guard intersection computation based on constrained zonotopes (see Sec. 2.2.1.9). Constrained zonotopes are closed under intersection. To this end, we first convert the reachable sets to constrained zonotopes and then intersect the reachable set with the guard sets. Finally, the union of all intersections is enclosed by oriented hyperrectangles, where the orientation is determined with the methods in Tab. 21. | |
| `hyperplaneMap` | The continuous dynamics are abstracted by constant flow, which allows to calculate the intersection with a hyperplane using a closed formula (guard mapping). | [101] |
| `pancake` | The dynamics of the system is scaled by the distance to the guard set so that the reachable set is very flat shortly before passing the guard set. It is then often possible to pass the guard set in a single time step. | [102] |
| `nondetGuard` | Guard intersection approach that works very well for nondeterministic guard sets. We first enclose all reachable sets that intersect the guard set with oriented hyperrectangles, where the orientation is determined using the methods in Tab. 21. Afterwards, we compute the intersection of the oriented hyperrectangles with the guard set. | |
| `levelSet` | The intersections between the reachable set and nonlinear guard sets are enclosed by polynomial zonotopes (see Sec. 2.2.1.5) | [103] |

Table 23: Supported combinations of guard sets and guard intersection methods. The shorthand `polytope` denotes all polytopic set representations, which are `interval`, `zonotope`, `polytope`, `conZonotope`, and `zonoBundle`.

| `options.guardIntersect` | `polytope` | `levelSet` |
|---|---|---|
| `polytope` | ✓ | ✗ |
| `zonoGirard` | ✗ | ✗ |
| `conZonotope` | ✓ | ✗ |
| `hyperplaneMap` | ✗ | ✗ |
| `pancake` | ✗ | ✗ |
| `nondetGuard` | ✓ | ✗ |
| `levelSet` | ✗ | ✓ |

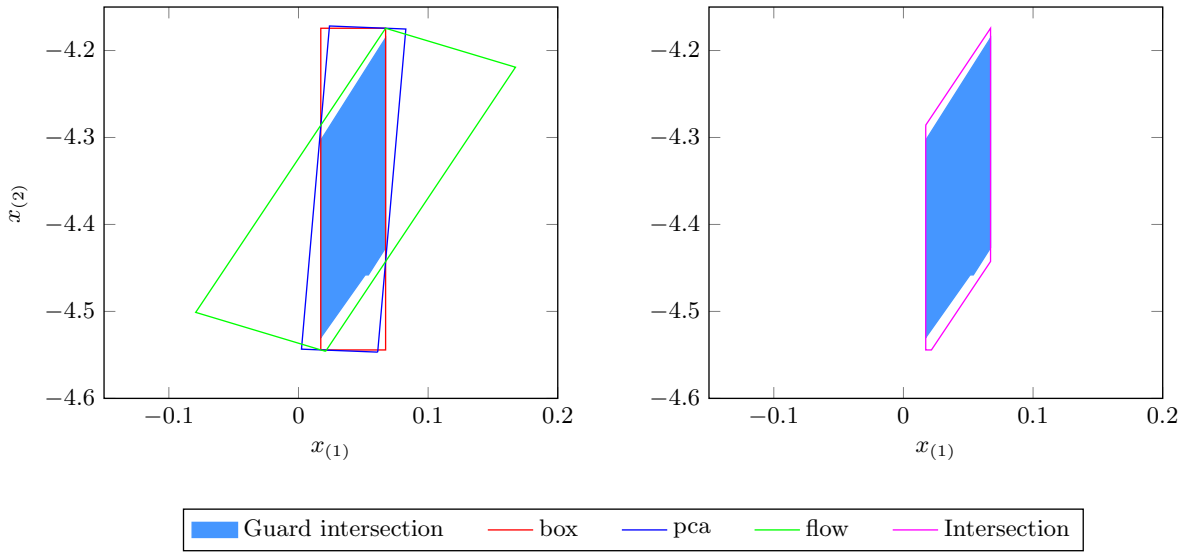



Figure 14: Enclosing hyperrectangles for different methods to obtain the orientation (left) and intersection between the hyperrectangles for all methods (right).

enters the corresponding part of the state space. Since usually only a small part of the state space is explored by simulation or reachability analysis, it is possible to significantly reduce the computational costs of the analysis if the system is modeled as a parallel hybrid automaton [105].

Parallel hybrid automata are implemented in CORA by the class `parallelHybridAutomaton`. An object of class `parallelHybridAutomaton` can be constructed as follows:

$$\texttt{pHA} = \texttt{parallelHybridAutomaton}(\texttt{components}, \texttt{inputBinds}),$$

with input arguments

- `components` – array of `hybridAutomaton` objects (see Sec. 4.3.1) containing all subcomponents of the system. Currently, only hybrid automata for which the continuous dynamics are modeled as a linear system (see Sec. 4.2.1) are supported.
- `inputBinds` – cell array containing matrices that describe the connections between the subcomponents. Each matrix has two columns: the first column represents the component the signal comes from and the second column the output number, e.g., $[2, 3]$ refers to output 3 of component 2. When an input to a component is also an input to the composed system, we use index 0, e.g., $[0, 1]$. For each input of the subcomponent, we specify a new row and the row number corresponds to the input index of the considered component.

For better illustration of the required information, we introduce the example presented in Fig. 15 consisting of three components. For the parallel hybrid automaton in this example, the input binds have to be specified as follows:

```
inputBinds{1} = [[0 2];[0 1];[2 1]]; % input connections for component 1
inputBinds{2} = [[0 1];[0 2]];       % input connections for component 2
inputBinds{3} = [[1 2];[2 2]];       % input connections for component 3
```

Let us briefly discuss the solution for component 1, which has three inputs and thus `inputBinds{1}` has three rows: The first input (first row) is the second input of the composed system; the second input is the first input of the composed system; and the third input is the first output of component 2.

Since the modeling of hybrid automata is tedious and error-prone, we provide a method to

read models of parallel hybrid automata using the SpaceEx format [65]. For modeling and modifying SpaceEx models, one can use the freely-available SpaceEx model editor downloadable from spaceex.imag.fr/download-6. Details on converting SpaceEx models to models as defined in this section can be found in Sec. 8.

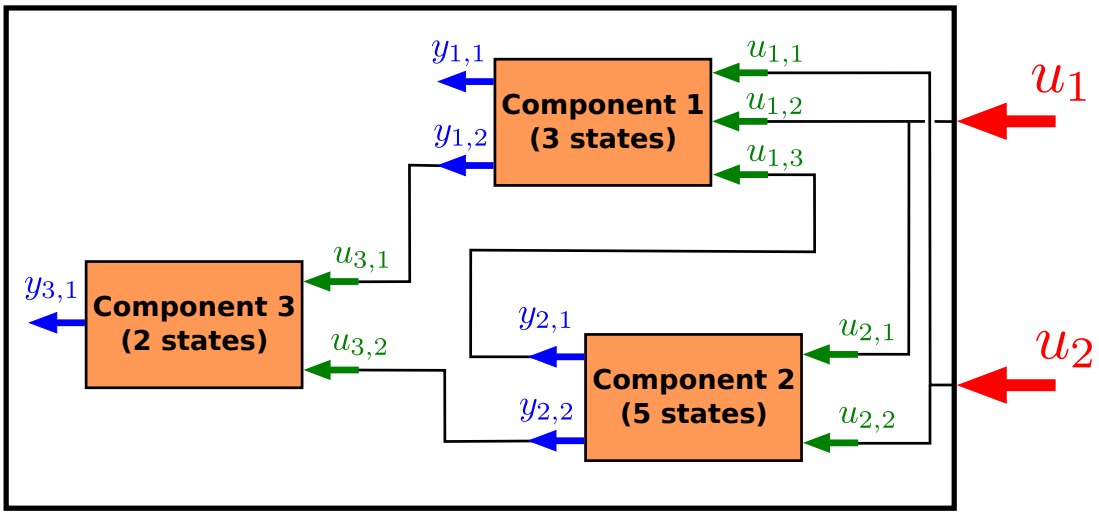



Figure 15: Example of a parallel hybrid automaton that consists of three subcomponents.

#### 4.3.2.1 Operation `reach`

The settings for reachability analysis are specified as fields of the struct `options` (see Sec. 4.1.1). For parallel hybrid automata, the settings are identical to the ones for hybrid automata (see Sec. 4.3.1.1).

The initial location `params.startLoc` and the final location `params.finalLoc` (see Sec. 4.1.1) are specified as a vector $l \in \mathbb{N}^s_{\geq 0}$, where each entry of the vector represents the index of the location for one of the $s$ subcomponents.

For the set of uncertain inputs specified by `params.U` (see Sec. 4.1.1), there exist two different cases for parallel hybrid automata:

1. The input set is identical for each component and location. In this case, a single set $\mathcal{U} \subset \mathbb{R}^m$ represented as a zonotope (see Sec. 2.2.1.1) is provided.
2. The input set is different for each component and location. In this case, `params.U` can be specified as a cell array, where each entry represents the input set for one component. Since each component can have multiple locations, the input set for each component is again a cell array whose entries represent the input sets for all locations. The input set for the overall system is then constructed on demand for each visited location according to

$$\mathcal{U} = \texttt{params.U}\{i_{(1)}\}\{l_{(i_{(1)})}\} \times \cdots \times \texttt{params.U}\{i_{(m)}\}\{l_{(i_{(m)})}\},$$

where the vector $l \in \mathbb{N}^s_{\geq 0}$ stores the index of the current location for all $s$ components, and the vector $i \in \mathbb{N}^m_{\geq 0}$ maps the input sets for the single components to the global input set. The vector $i$ can be specified with an additional setting `params.inputCompMap` $= i$.

# 5 Abstraction to Discrete Systems

## 5.1 State Space Partitioning

It is sometimes useful to partition the state space into cells, for instance, when abstracting a continuous stochastic system by a discrete stochastic system. CORA supports axis-aligned partitioning using the class `partition`.

We mainly support the following methods for partitions:

- `cellCenter` – returns a cell array of cell center positions of the partition segments whose indices are given as input.
- `cellIndices` – returns cell indices given a set of cell coordinates.
- `cellIntervals` – returns a cell array of interval objects corresponding to the cells specified as input.
- `cellPolytopes` – returns polytopes of selected cells.
- `cellSegments` – returns cell coordinates given a set of cell indices.
- `cellZonotopes` – returns zonotopes of selected cells.
- `display` – displays the parameters of the partition in the MATLAB workspace.
- `exactIntersectingCells` – finds the exact cells of the partition that intersect a set P, and the proportion of P that is in each cell.
- `intersectingCells` – returns the cells possibly intersecting with a continuous set, over-approximatively, by over-approximating the convex set as a multidimensional interval.
- `nrOfCells` – returns the number of cells of the partition.
- `findSegments` – returns segment indices intersecting with a given multidimensional interval.
- `nrOfStates` – returns the number of discrete states of the partition.
- `partition` – constructor of the class.
- `plot` – plots the partition.

## 5.2 Abstraction to Markov Chains

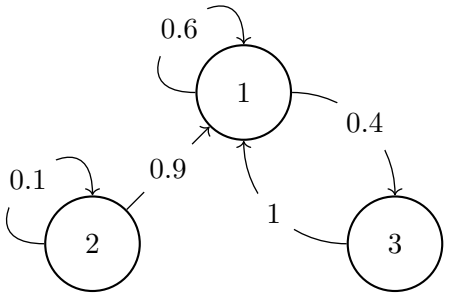
The main idea of the Markov chain abstraction is to analyze a dynamic system probabilistically by a Markov chain instead of making use of the original system dynamics. The Markov chain abstraction has to be performed so that it approximates the behavior of the original system with appropriate accuracy. The abstraction can be applied to both continuous and hybrid systems. Since Markov chains are stochastic systems with a discrete state space, the continuous state space of the original state and input space has to be discretized for the abstraction as presented in Sec. 5.1. This implies that the number of states of the Markov chain grows exponentially with the dimension of the continuous state space. Thus, the presented abstraction is only applicable to low-dimensional systems of typically up to 4 continuous state variables.

The following definition of Markov chains is adapted from [106]: A discrete time Markov chain $MC = (Y, \hat{p}^0, \Phi)$ consists of

- The countable set of locations $Y \subset \mathbb{N}_{>0}$.

- The initial probability $\hat{p}_i^0 = P(\mathbf{z}(0) = i)$, with random state $\mathbf{z} : \Omega \to Y$, where $\Omega$ is the set of elementary events and $P()$ is an operator determining the probability of an event.
- The transition matrix $\Phi_{ij} = P(\mathbf{z}(k+1) = i | \mathbf{z}(k) = j)$ so that $\hat{p}(k+1) = \Phi\hat{p}(k)$.

Clearly, the Markov chain fulfills the Markov property, i.e., the probability distribution of the future time step $\hat{p}(k+1)$ depends only on the probability distribution of the current time step $\hat{p}(k)$. If a process does not fulfill this property, one can always augment the discrete state space by states of previous time steps, allowing the construction of a Markov chain with the new state $\mathbf{z}^*(k)^T = \left[\mathbf{z}(k)^T, \mathbf{z}(k-1)^T, \mathbf{z}(k-2)^T, \ldots\right]$. An example of a Markov chain is visualized in Fig. 16 by a graph whose nodes represent the states $1, 2, 3$ and whose labeled arrows represent the transition probabilities $\Phi_{ij}$ from state $j$ to $i$.




$$\Phi = \begin{bmatrix} 0.6 & 0.9 & 1 \\ 0 & 0.1 & 0 \\ 0.4 & 0 & 0 \end{bmatrix}$$

Figure 16: Exemplary Markov chain with 3 states.

The relation of the discrete time step $k$ and the continuous time is established by introducing the time increment $\tau \in \mathbb{R}^+$ after which the Markov chain is updated according to the transition matrix $\Phi$. Thus, the continuous time at time step $k$ is $t_k = k \cdot \tau$. The generation of a Markov chain from continuous dynamics is performed as described in [33, Sec. 4.3].

We mainly support the following methods for Markov chains:

- `build` – builds the transition matrices of the Markov chains using simulation.
- `build_reach` – builds the transition matrices of the Markov chains using reachability analysis.
- `convertTransitionMatrix` – converts the transition matrix of a Markov chain so that it can be used for an optimized update as presented in [107].
- `markovchain` – constructor of the class.
- `plot` – generates 3 plots of a Markov chain: 1. sample trajectories; 2. reachable cells for the final time; 3. reachable cells for the time interval.
- `plot_reach` – generates 3 plots of a Markov chain: 1. continuous reachable set together with sample trajectories; 2. reachable cells for the final time; 3. reachable cells for the time interval.
- `plotP` – plots the 2D probability distribution of a Markov chain.

## 5.3 Stochastic Prediction of Road Vehicles

An important application of abstracting hybrid dynamics to Markov chains is the probabilistic prediction of traffic participants as presented in, e.g., [107, 108]. The probabilistic information

allows one not only to check if a planned path of an autonomous vehicle may result in a crash, but also with which probability. Consequently, possible driving strategies of autonomous cars can be evaluated according to their safety. Traffic participants are abstracted by Markov chains as presented in Sec. 5.2. There are three properties which are in favor of the Markov chain approach: The approach can handle the hybrid dynamics of traffic participants, the number of continuous state variables (position and velocity) is low, and Markov chains are computationally inexpensive when they are not too large.

We provide all numerical examples presented in [33, Sec. 5]. Please note that the code is not as clean as for the core CORA classes since this part of the code is not a foundation for other implementations, but rather a demonstration of probabilistic predictions of road traffic. To replicate the braking scenario in [33, Sec. 5], perform the following steps:

1. Run `/discrDynamics/ProbOccupancyPrediction/intersection/start_intersectionDatabase` to obtain an intersection database. The result is a structure `fArray`. Caution: Executing this function can take several hours.
2. Run `start_carReach` to compute the Markov chain of a traffic participant. You have to select the corresponding `fArray` file to make sure that the segment length of the path is consistent. The type of traffic participant is exchanged by exchanging the loaded hybrid automaton model, e.g., to load the bicycle model use `[HA,...] = initBicycle(fArray.segmentLength)`. Finally, save the resulting probabilistic model. Caution: Executing this function can take several hours.
3. (optional) Instead of computing the Markov chain by simulations, one can compute it using reachability analysis by using `carReach_reach`.
4. Select the scenario; each scenario requires to load a certain amount of MC models. The following set of scenarios are currently available:
   - braking
   - intersectionCrossing
   - knownBehavior
   - laneChange
   - merging
   - overtaking
   - straightVScurved

As an example, the outcome of the braking scenario is described subsequently. The interaction between vehicles in a lane is demonstrated for 3 cars driving one after the other. The cars are denoted by the capital letters $A$, $B$, and $C$, where $A$ is the first and $C$ the last vehicle in driving direction. Vehicle $A$ is not computed based on a Markov chain, but predicted with a constant velocity of 3 m/s so that the faster vehicles $B$ and $C$ are forced to brake. The probability distributions for a selected time interval is plotted in Fig. 17. For visualization reasons, the position distributions are plotted in separate plots, although the vehicles drive in the same lane. Darker regions indicate high probability, while brighter regions represent areas of low probability. In order to improve the visualization, the colors are separately normalized for each vehicle.

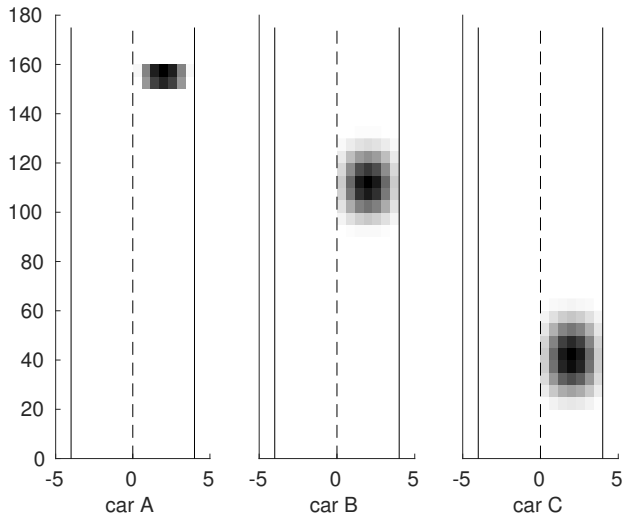



Figure 17: Probabilistic occupancy prediction of the braking scenario.

# 6 Neural Networks

In this section, we describe the capabilities of CORA to formally verify existing neural networks. Additionally, one can train robust neural networks in CORA by integrating formal methods into the training process. Please also install the required toolboxes described in Sec. 1.3 when using neural networks in CORA.

## 6.1 Neural Networks in CORA

Neural networks are represented in CORA as objects of the class `neuralNetwork`. They can be initialized directly in CORA or imported from a file using various formats, e.g., ONNX. Currently, CORA mainly supports feed-forward neural networks (see Fig. 18) with various activation functions.

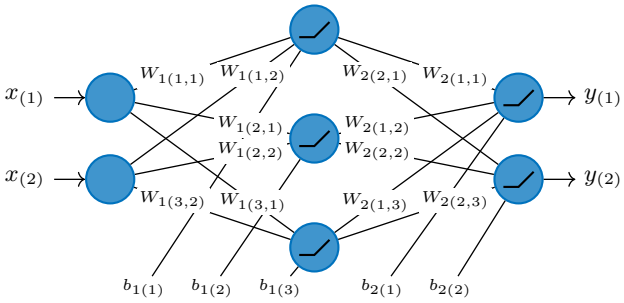



Figure 18: Feed-forward neural network with $\kappa = 2$ layers.

An object of class `neuralNetwork` can be constructed as follows:

```
nn = neuralNetwork(layers),
```

where `layers` is a cell array of length $\kappa$ filled with objects of type `nnLayer`. For a network $\Phi\colon \mathbb{R}^{n_0} \to \mathbb{R}^{n_\kappa}$ with input $x \in \mathbb{R}^{n_0}$ and output $y \in \mathbb{R}^{n_\kappa}$, each layer $k \in [\kappa]$ computes $h_k = L_k\,(h_k - 1) \in \mathbb{R}^{n_k}$, where $h_0 = x$ and $y = h_\kappa$ and thus obtaining $y = \Phi(x)$. Currently, the following layers are supported in CORA:

- `nnLinearLayer(W, b)` Linear layer computing $L_k^{\text{LIN}}\,(h_{k-1}) = W_k h_{k-1} + b_k$ with weight matrix $W \in \mathbb{R}^{n_k \times n_{k-1}}$ and bias $b \in \mathbb{R}^{n_k}$ (Fig. 18).
- `nnActivationLayer()` Abstract activation layer computing $L_k^{\text{ACT}}\,(h_{k-1}) = \phi(h_{k-1})$, where $\phi$ is the corresponding element-wise activation function.
- `nnLeakyReLULayer()` `nnActivationLayer()` with $\phi(h_{k-1}) = \max(\alpha h_{k-1}, h_{k-1})$, $\alpha \in \mathbb{R}$ with default $\alpha = 0.01$.
- `nnReLULayer()` `nnLeakyReLULayer()` with $\alpha = 0$, thus $\phi(h_{k-1}) = \max(0, h_{k-1})$.
- `nnSigmoidLayer()` `nnActivationLayer()` with $\phi(h_{k-1}) = \text{sigmoid}(h_{k-1})$.
- `nnTanhLayer()` `nnActivationLayer()` with $\phi(h_{k-1}) = \tanh(h_{k-1})$.

Let us demonstrate the `neuralNetwork` class by the following example, which constructs a neural network with the same structure as the one shown in Fig. 18:

```
% init weight and bias
W1 = rand(3,2); b1 = rand(3,1);
W2 = rand(2,3); b2 = rand(2,1);

% neural network
nn = neuralNetwork({
    nnLinearLayer(W1, b1);
    nnReLULayer();
    nnLinearLayer(W2, b2);
    nnReLULayer();
})
```

While the example initializes a random neural network, one can also import neural networks into CORA from various common neural network formats. CORA currently supports ONNX, NNet, YML, Sherlock, and the conversion for neural networks coming from the Deep Learning Toolbox. Please note that additional toolboxes are required for the ONNX import (see Sec. 1.3).

Please visit `./cora/nn` for further details.

## 6.2 Formal Verification of Neural Networks

CORA enables the formal verification of neural networks. This refers to the problem of given a neural networks $\Phi\colon \mathbb{R}^{n_0} \to \mathbb{R}^{n_\kappa}$, an input set $\mathcal{X} \subset \mathbb{R}^{n_0}$, and some unsafe set $\mathcal{S} \subset \mathbb{R}^{n_\kappa}$ in the output space of the neural network, we want to verify that it holds

$$\Phi(\mathcal{X}) \cap \mathcal{S} = \emptyset. \tag{52}$$

As it is in general not feasible to compute the exact output set $\mathcal{Y}^* = \Phi(\mathcal{X})$ [109], CORA usually computes an enclosure of the output set $\mathcal{Y} \supseteq \mathcal{Y}^*$. This is done using reachability analysis.

CORA primarily implements the approaches described in [97,98] for the reachability analysis on neural networks. The class `neuralNetwork` provides the function `evaluate`, that can be used to compute the output of a neural network for a single point or a set of possible inputs:

```
res = nn.evaluate(x,options,idxLayer),
res = nn.evaluate(X,options,idxLayer),
```

where $\mathtt{x} \in \mathbb{R}^{n_0 \times N}$ is an array of $N$ input points and $\mathtt{X} \subset \mathbb{R}^{n_0}$ is an input set. Currently, CORA supports zonotopes, polynomial zonotopes, interval, Taylor models, and constraint zonotopes as input sets to the neural network. A few settings are available to control the set propagation through the neural network. These are stored in the algorithm options struct for neural networks `options.nn`. We list the most important settings below, further information can be found in the documentation of the `evaluate` function:

- `.poly_method` string describing the approximation method of nonlinear functions: `'regression'`, `'singh'`, and `'taylor'`.
- `.num_generators` maximal number of generators of the polynomial zonotope. If this number is exceeded, an order reduction method is executed.

The following code shows an exemplary set propagation through the previously defined neural network:

```
% input set
c = [4;4];
G = [2 1 2; 0 2 2];
E = [1 0 3;0 1 1];
GI = [];
X = polyZonotope(c,G,GI,E);

% settings
options = struct;
options.nn.poly_method = 'regression';
options.nn.num_generators = 1000;

% evaluation
Y = nn.evaluate(X, options);
```

CORA can also automatically verify given specifications using the `verify` function:

```
[res,x_,y_] = nn.verify(x,r,A,b,varargin),
```

where `x, r` is the center of the input and the perturbation radius $\epsilon$ of the input set $\mathcal{X}$, and `A, b` describe the specification $A\tilde{y} + b \leq 0$ with $\tilde{y} = \Phi(\tilde{x})$, $\tilde{x} \in \mathcal{X}$. The output variable `res` is `true` if the specification is fulfilled $\forall \tilde{x} \in \mathcal{X}$, `false` if a counterexample was found, and is empty otherwise. In case a counterexample was found, `x_, y_` contains the respective point of the input set and output space, respectively.

Further examples can be found at `./cora/examples/nn`.

### 6.3 Neural Networks as Controllers

CORA also allows the verification of dynamic systems where neural networks are used to control the system dynamics. Please visit Sec. 4.2.11 for more information.

Examples can be found at `./cora/examples/contDynamics/neurNetContrSys`.

### 6.4 Training Verifiably Robust Neural Networks

CORA also enables the training of verifiably robust neural networks both in the supervised [3] and in the reinforcement learning [4] setting. This is realized by integrating reachability analysis into the training process. In particular, we propagate zonotopes through a neural network and compute a set-based gradient based on the computed output set $\mathcal{Y}$, target $t$, and loss function $E$. This is done efficiently in CORA by propagating the sets batch-wise through the network and every computation can be done on a GPU.

The set-based loss function is defined as

$$E_{\text{set}}(t, \mathcal{Y}) = (1 - \tau) E_{\text{point}}(t, \mathcal{Y}) + \tau E_{\text{volume}}(\mathcal{Y}), \tag{53}$$

where $E_{\text{point}}$ refers to the standard (point-based) loss, e.g., mean squared error or cross-entropy loss, and $E_{\text{volume}}$ to a volume loss to reduce the size of the output set. The parameter $\tau \in [0, 1]$ is a hyperparameter to weight the two terms. The process of this set-based training is schematically shown in Fig. 19. The resulting networks are verifiably more robust against input perturbations as these were considered during the training process.

In the supervised setting, this is done via the function `train`:

```
[loss,trainTime] = nn.train(trainX, trainT, valX, valT, options, verbose),
```

where `trainX, trainT, valX, valT` are the training and validation dataset, respectively, algorithm options for training are stored in `options.nn.train`, and `verbose` is a logical flag for

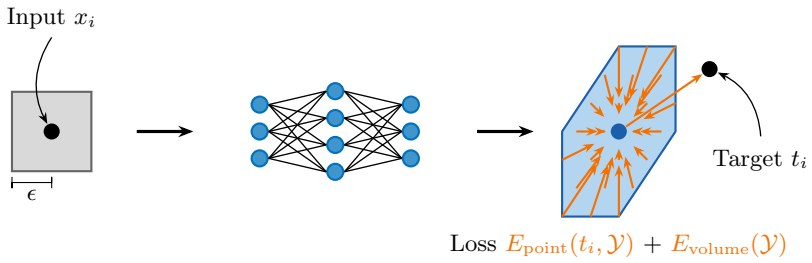



Figure 19: Set-based training of a neural network to make it verifiably robust against input perturbations.

verbose outputs to the command window during training. The training options specify all hyperparameters for training, e.g., the number of epochs, the optimizer, the batch size, and the loss. See the code below for an example and further details can be found in the documentation of the `train` function.

```
% Initialize neural network
nn = neuralNetwork(layers);
nn.initWeights();

% Train neural network
options.nn = struct(...
    'use_approx_error',true,...
    'poly_method','bounds',...
    'train',struct( ...
        'optim',nnSGDOptimizer(lr,beta),...
        'max_epoch',maxEpoch,...
        'mini_batch_size',miniBatchSize,...
        'loss','mse',...
        'method','set',...
        'noise',0.5,...
        'input_space_inf',xl,...
        'input_space_sup',xu,...
        'tau',0.5,...
        'volume_heuristic','f-radius',...
        'zonotope_weight_update','sum',...
        'num_approx_err',inf,...
        'init_gens','l_inf',...
        'num_init_gens',inf...
    )...
);
loss = nn.train(trainX,trainY,valX,valY,options,true);
```

Please visit `./cora/nn` for details and `./cora/examples/nn` for examples.

CORA also enables the training of verifiably robust agents in an actor-critic settings, which is depicted in Fig. 20: After defining a control environment specifying the dynamic of the model, and actor $\mu_\phi$, critic $Q_\theta$ networks with parameters $\phi$, $\theta$, respectively, one can also use set-based training in reinforcement learning using CORA.

The respective classes in CORA are `ctrlEnvironment`, `actor`, and `critic`, where both, agents based on the DDPG (`DDPGagent`) and TD3 (`TD3agent`) algorithm can be trained. The options for the agent and critic are as described above for the supervised settings, which are stored in `options.rl.actor.nn.train` and `options.rl.critic.nn.train`, respectively. Additional options are available for the environment, such as initial set and time steps, which are stored in

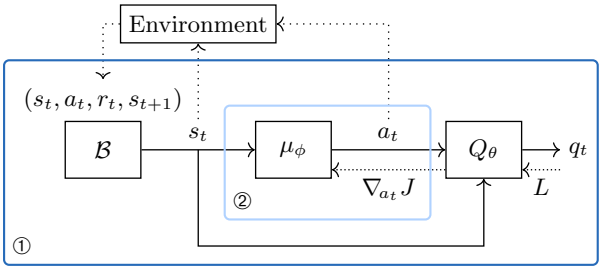



Figure 20: Illustration of the structure of the deep deterministic policy gradient algorithm: ① and ② show the components that are augmented through set-based training.

`options.rl.env`. Forther details can be found in the documentation of the respective classes. Please note that either the actor and critic can be trained set-based (① in Fig. 20), or only the actor (② in Fig. 20).

Please visit `./cora/nn/rl` for details and `./cora/examples/nn` for examples.

# 7 Additional Functionality

In this section, we describe additional functionality implemented in CORA.

## 7.1 Class `reachSet`

Reachable sets are stored as objects of class `reachSet`. This class implements several useful methods that make it very convenient to handle the resulting reachable sets.

An object of class `reachSet` can be constructed as follows:

$$\begin{aligned}
&\texttt{R} = \texttt{reachSet}(\texttt{timePoint}),\\
&\texttt{R} = \texttt{reachSet}(\texttt{timePoint}, \texttt{parent}),\\
&\texttt{R} = \texttt{reachSet}(\texttt{timePoint}, \texttt{parent}, \texttt{loc}),\\
&\texttt{R} = \texttt{reachSet}(\texttt{timePoint}, \texttt{timeInt}),\\
&\texttt{R} = \texttt{reachSet}(\texttt{timePoint}, \texttt{timeInt}, \texttt{parent}),\\
&\texttt{R} = \texttt{reachSet}(\texttt{timePoint}, \texttt{timeInt}, \texttt{parent}, \texttt{loc}),
\end{aligned}$$

with input arguments

- `timePoint` struct with fields `.set` and `.time` storing reachable sets of time points.
- `timeInt` struct with fields `.set`, `.time`, and `.algebraic` (`nonlinDASys` only, see Sec. 4.2.10) storing reachable sets of time intervals.
- `parent` index of the parent reachable set.
- `loc` index of the location (see Sec. 4.3) to which the reachable set belongs (hybrid systems only).

The reachable set can consist of multiple strands as visualized in Fig. 21. New strands are created at location changes for hybrid systems, if reachable sets are split, and if reachable sets are united. For the reachable set shown in Fig. 21, the corresponding `reachSet` object is as follows:

```
R =

5x1 reachSet array:

  timePoint
  timeInterval
  parent
  loc
```

```
R(1)

 reachSet with properties:

     timePoint: [1x1 struct]
  timeInterval: [1x1 struct]
        parent: 0
           loc: 1
```

```
R(2)

 reachSet with properties:

     timePoint: [1x1 struct]
  timeInterval: [1x1 struct]
        parent: 1
           loc: 2
```

```
R(3)

 reachSet with properties:

     timePoint: [1x1 struct]
  timeInterval: [1x1 struct]
        parent: 2
           loc: 2
```

```
R(4)

 reachSet with properties:

     timePoint: [1x1 struct]
  timeInterval: [1x1 struct]
        parent: 2
           loc: 2
```

```
R(5)

 reachSet with properties:

     timePoint: [1x1 struct]
  timeInterval: [1x1 struct]
        parent: [3,4]
           loc: 2
```

Next, we explain the most common methods for the class `reachSet` in detail.

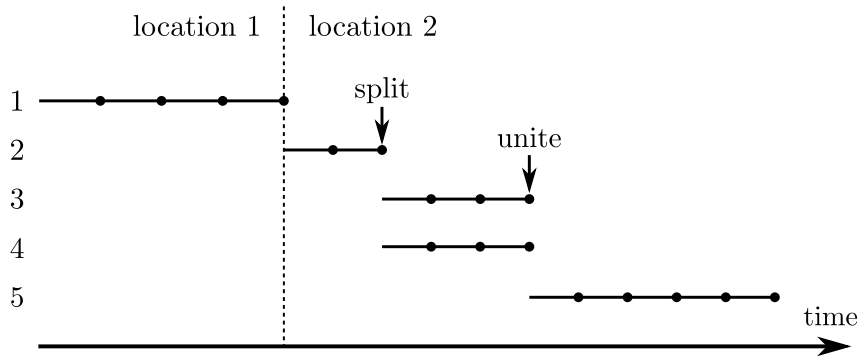



Figure 21: Example demonstrating the different strands of the reachable set.

### 7.1.1 `add`

The method `add` adds a reachable set to another one:

$$\texttt{R} = \texttt{add}(\texttt{R1}, \texttt{R2}),$$
$$\texttt{R} = \texttt{add}(\texttt{R1}, \texttt{R2}, \texttt{parent}),$$

where `R1` and `R2` are both objects of class `reachSet`, and `parent` is the index of the parent for the root element of `R2`. Adding reachable sets is for example useful if the overall reachable set is computed in multiple sequences.

### 7.1.2 `find`

The method `find` returns all reachable sets that satisfy the specified condition:

$$\texttt{res} = \texttt{find}(\texttt{R}, \texttt{prop}, \texttt{val}),$$

where `R` is an object of class `reachSet`, `prop` is a string specifying the property for the condition, `val` is the desired value of the property, and `res` is an object of class `reachSet` containing all reachable sets that satisfy the property. Currently, the following values for `prop` are supported:

- `'location'`: find all reachable sets that correspond to the specified location.
- `'parent'`: find all reachable sets with the specified parent.
- `'time'`: find all reachable sets that correspond to the specified time interval.

### 7.1.3 `plot`

The method `plot` visualizes a two-dimensional projection of the boundary of reachable set for time intervals:

$$\texttt{han} = \texttt{plot}(\texttt{R}),$$
$$\texttt{han} = \texttt{plot}(\texttt{R}, \texttt{dim}),$$
$$\texttt{han} = \texttt{plot}(\texttt{R}, \texttt{dim}, \texttt{linespec}),$$
$$\texttt{han} = \texttt{plot}(\texttt{R}, \texttt{dims}, \texttt{linespec}, \texttt{namevaluepairs}),$$

where `R` is an object of class `reachSet`, `han` is a handle to the plotted MATLAB graphics object, and the additional input arguments are defined as

- `dims`: Integer vector $\texttt{dims} \in \mathbb{N}^2_{\leq n}$ specifying the dimensions for which the projection is visualized (default value: `dims = [1 2]`).

- `linespec`: (optional) line specifications, e.g., `'--*r'`, as supported by MATLAB[19].
- `namevaluepairs`: (optional) further specifications as name-value pairs, e.g., `'LineWidth',2` and `'FaceColor',[.5 .5 .5]`, as supported by MATLAB. If the plot is not filled, these are the built-in Line Properties[20], if the plot is filled, they correspond to the Patch Properties[21].

The following name-value pairs enhance the built-in functionalities:

- `'Order'`: zonotope order for plotting. If provided, the zonotope order is reduced to the given order before the set is plotted.
- `'Splits'`: number of splits applied to refine the plotted over-approximation of polynomial zonotopes (polynomial zonotopes only, see Sec. 2.2.1.5).
- `'Unify'`: If the name-value pair `'Unify',true` is passed the union of all reachable sets is computed to avoid overlapping regions in the plot. The resulting figure then usually requires significantly less storage space.

For discrete-time systems (see Sec. 4.2.3 and Sec. 4.2.8), the reachable set at time points is visualized since there exists no reachable set for time intervals.

### 7.1.4 `plotOverTime`

The method `plotOverTime` visualizes a one-dimensional projection of the reachable set of time intervals over time:

$$\begin{aligned}
&\texttt{han} = \texttt{plotOverTime}(\texttt{R}),\\
&\texttt{han} = \texttt{plotOverTime}(\texttt{R}, \texttt{dims}),\\
&\texttt{han} = \texttt{plotOverTime}(\texttt{R}, \texttt{dims}, \texttt{linespec}),\\
&\texttt{han} = \texttt{plotOverTime}(\texttt{R}, \texttt{dims}, \texttt{linespec}, \texttt{namevaluepairs}),
\end{aligned}$$

where `R` is an object of class `reachSet`, `han` is a handle to the plotted MATLAB graphics object, and the additional input arguments are defined as

- `dims`: Integer vector $\texttt{dims} \in \mathbb{N}_{\leq n}$ specifying the dimensions for which the projection is visualized (default value: `dim = 1`).
- `linespec`: (optional) line specifications, e.g., `'--*r'`, as supported by MATLAB[22].
- `namevaluepairs`: (optional) further specifications as name-value pairs, e.g., `'LineWidth',2` and `'FaceColor',[.5 .5 .5]`, as supported by MATLAB. They correspond to the Patch Properties[23].

The following name-value pairs enhance the built-in functionalities:

- `'Unify'`: If the name-value pair `'Unify',true` is passed the union of all reachable sets is computed to avoid overlapping regions in the plot. The resulting figure then usually requires significantly less storage space.

For discrete-time systems (see Sec. 4.2.3 and Sec. 4.2.8), the reachable set at time points is visualized since there exists no reachable set for time intervals.

[19] https://de.mathworks.com/help/matlab/ref/linespec.html
[20] https://de.mathworks.com/help/matlab/ref/matlab.graphics.chart.primitive.line-properties.html
[21] https://de.mathworks.com/help/matlab/ref/matlab.graphics.primitive.patch-properties.html
[22] https://de.mathworks.com/help/matlab/ref/linespec.html
[23] https://de.mathworks.com/help/matlab/ref/matlab.graphics.primitive.patch-properties.html

### 7.1.5 query

The method `query` returns the value of a certain property of an object of class `reachSet`:

$$\texttt{val} = \texttt{query}(\texttt{R}, \texttt{prop}),$$

where `R` is an object of class `reachSet`, `prop` is a string specifying the property of interest, and `val` is the value of the property. Currently, the following values for `prop` are supported:

- `'reachSet'`: returns all reachable sets of time intervals as a cell array.
- `'reachSetTimePoint'`: returns all reachable sets at points in time as a cell array.
- `'finalSet'`: returns the last time-point reachable set.
- `'tVec'`: returns the vector of time step sizes (only supported for one single strand).

## 7.2 Class `simResult`

The results of simulations are stored in CORA as objects of the class `simResult`, which provides several methods to easily visualize the simulated trajectories. An object of class `simResult` can be constructed as follows:

$$\begin{aligned}\texttt{simRes} &= \texttt{simResult}(\texttt{x}, \texttt{t}),\\ \texttt{simRes} &= \texttt{simResult}(\texttt{x}, \texttt{t}, \texttt{loc}),\end{aligned}$$

with input arguments

- `x` cell array storing the states of the simulated trajectories.
- `t` cell array storing the time points for the simulated trajectories.
- `loc` cell array storing the indices of the locations for the simulated trajectories (hybrid systems only).

Next, we explain the methods of the class `simResult` in detail.

### 7.2.1 add

The method `add` combines two `simResult` objects `simRes1` and `simRes2`:

$$\texttt{simRes} = \texttt{add}(\texttt{simRes1}, \texttt{simRes2}).$$

### 7.2.2 plot

The method `plot` visualizes a two-dimensional projection of the obtained trajectories:

$$\begin{aligned}\texttt{han} &= \texttt{plot}(\texttt{simRes}),\\ \texttt{han} &= \texttt{plot}(\texttt{simRes}, \texttt{dims}),\\ \texttt{han} &= \texttt{plot}(\texttt{simRes}, \texttt{dims}, \texttt{linespec}),\\ \texttt{han} &= \texttt{plot}(\texttt{simRes}, \texttt{dims}, \texttt{namevaluepairs}),\end{aligned}$$

where `simRes` is an object of class `simResult`, `han` is a handle to the plotted MATLAB graphics object, and the additional input arguments are defined as

- `dims`: Integer vector $\texttt{dims} \in \mathbb{N}^2_{\leq n}$ specifying the dimensions for which the projection is visualized (default value: `dims = [1 2]`).

- `linespec`: (optional) line specifications, e.g., `'--*r'`, as supported by MATLAB[24] (default value: `linespec = 'b'`).
- `namevaluepairs`: (optional) further specifications as name-value pairs, e.g., `'LineWidth',2` and `'MarkerSize',1.5`, as supported by MATLAB. They correspond to the Line Properties[25].

### 7.2.3 `plotOverTime`

The method `plotOverTime` visualizes a one-dimensional projection of the simulated trajectories over time:

$$\begin{aligned}
&\texttt{han} = \texttt{plotOverTime}(\texttt{simRes}),\\
&\texttt{han} = \texttt{plotOverTime}(\texttt{simRes}, \texttt{dims}),\\
&\texttt{han} = \texttt{plotOverTime}(\texttt{simRes}, \texttt{dims}, \texttt{linespec}),\\
&\texttt{han} = \texttt{plotOverTime}(\texttt{simRes}, \texttt{dims}, \texttt{namevaluepairs}),
\end{aligned}$$

where `simRes` is an object of class `simResult`, `han` is a handle to the plotted MATLAB graphics object, and the additional input arguments are defined as

- `dims`: Integer vector $\texttt{dims} \in \mathbb{N}_{\leq n}$ specifying the dimensions for which the projection is visualized (default value: `dims = 1`).
- `linespec`: (optional) line specifications, e.g., `'--*r'`, as supported by MATLAB[26].
- `namevaluepairs`: (optional) further specifications as name-value pairs, e.g., `'LineWidth',2` and `'MarkerSize',1.5`, as supported by MATLAB. They correspond to the Line Properties[27].

## 7.3 Class `specification`

The class `specification` allows one to define specifications that a system has to satisfy (see Sec. 4.1.1). If specifications are provided, reachability analysis terminates as soon as a specification is violated. An object of class `specification` can be constructed as follows (note that

[24] https://de.mathworks.com/help/matlab/ref/linespec.html
[25] https://de.mathworks.com/help/matlab/ref/matlab.graphics.chart.primitive.line-properties.html
[26] https://de.mathworks.com/help/matlab/ref/linespec.html
[27] https://de.mathworks.com/help/matlab/ref/matlab.graphics.primitive.patch-properties.html

$\mathcal{S}$ can be replaced by `list`):

$$
\begin{aligned}
&\texttt{spec} = \texttt{specification}(\mathcal{S}),\\
&\texttt{spec} = \texttt{specification}(\mathcal{S}, \texttt{type}),\\
&\texttt{spec} = \texttt{specification}(\mathcal{S}, \texttt{location}),\\
&\texttt{spec} = \texttt{specification}(\mathcal{S}, \texttt{type}, \texttt{location}),\\
&\texttt{spec} = \texttt{specification}(\mathcal{S}, \texttt{type}, \texttt{time}),\\
&\texttt{spec} = \texttt{specification}(\mathcal{S}, \texttt{type}, \texttt{location}, \texttt{time}),\\
&\texttt{spec} = \texttt{specification}(\mathcal{S}, \texttt{type}, \texttt{time}, \texttt{location}),\\
&\texttt{spec} = \texttt{specification}(\phi, \texttt{'logic'}),\\
&\texttt{spec} = \texttt{specification}(\texttt{func}, \texttt{'custom'}),\\
&\texttt{spec} = \texttt{specification}(\texttt{func}, \texttt{'custom'}, \texttt{time}),\\
&\texttt{spec} = \texttt{specification}(\texttt{func}, \texttt{'custom'}, \texttt{location}),\\
&\texttt{spec} = \texttt{specification}(\texttt{func}, \texttt{'custom'}, \texttt{time}, \texttt{location}),\\
&\texttt{spec} = \texttt{specification}(\texttt{func}, \texttt{'custom'}, \texttt{location}, \texttt{time}),
\end{aligned}
$$

where the input arguments are defined as follows:

- $\mathcal{S}$ set which defines the specification represented by one of the set representations in Sec. 2.2.
- `list` cell array storing the sets which define the specifications. Useful for constructing multiple specifications at once.
- `type` string specifying the type of the specifications. Supported types are `'unsafeSet'`, `'safeSet'`, `'invariant'`, and `'custom'`.
- `location` for hybrid automata (see Sec. 4.3.1): double array specifying in which location of a hybrid automaton the specification is active, can also be set to `[]` meaning active in all locations (default); for parallel hybrid automata (see Sec. 4.3.2): cell-array of double arrays specifying list of active locations for each component of the hybrid automaton
- `time` time interval in which the specification is valid specified as an object of class interval (see Sec. 2.2.1.2). The default value is the empty interval, which stands for valid at all times.
- $\phi$ temporal logic specification represented as an object of class `stl` (see Sec. 7.9).
- `func` function handle to a function $f(\mathcal{R})$ that takes the current reachable set $\mathcal{R}$ for time intervals as an input argument and returns `true` if the custom specification is satisfied, and `false` otherwise.

Let us denote the reachable set at time $t$ as $\mathcal{R}(t)$. The different types of specifications are defined as follows:

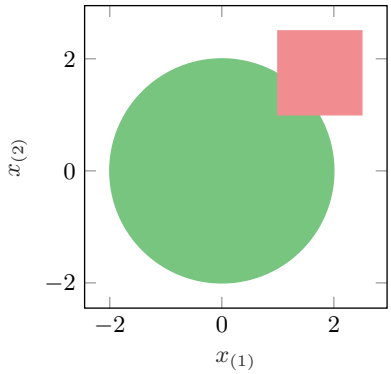
$$
\begin{aligned}
&\texttt{'unsafeSet'}: && \forall t \in [t_0, t_f]: \ \mathcal{R}(t) \cap \mathcal{S} = \emptyset\\
&\texttt{'safeSet'}: && \forall t \in [t_0, t_f]: \ \mathcal{R}(t) \subseteq \mathcal{S}\\
&\texttt{'invariant'}^{28}: && \forall t \in [t_0, t_f]: \ \mathcal{R}(t) \cap \mathcal{S} \neq \emptyset\\
&\texttt{'logic'}: && \forall \xi(t) \in \mathcal{R}(t): \ \xi(t) \models \phi\\
&\texttt{'custom'}: && \forall t \in [t_0, t_f]: \ f(\mathcal{R}(t)) = 1,
\end{aligned}
$$

where $t_0$ is the initial and $t_f$ the final time for the reachable set computation. It is also possible

[28] Please note that this specification does not check for invariants as defined in [110], but whether a reachable set is still within an invariant $\mathcal{S}$ as specified for hybrid systems.

to combine mutliple specifications using the method `add` (see Sec. 7.3.1). Let us demonstrate the construction of a specification by an example:

```
% first specification
S = ellipsoid(diag([4,4]));
spec1 = specification(S,'safeSet');

% second specification
S = interval([1;1],[2.5;2.5]);
spec2 = specification(S,'unsafeSet');

% combination of both specifications
spec = add(spec1,spec2);
```




Next, we explain the methods of class `specification` in detail.

### 7.3.1 add

The method `add` unites two specifications:

$$\texttt{spec} = \texttt{add}(\texttt{spec1}, \texttt{spec2}),$$

where `spec1` and `spec2` are both objects of class `specification`. The specifications defined by `spec1` and `spec2` both have to be satisfied for the resulting specification `spec` to be satisfied.

### 7.3.2 check

The method `check` checks if a set $\mathcal{S} \subset \mathbb{R}^n$ satisfies the specification defined by the object `spec` of class `specification`:

$$\texttt{res} = \texttt{check}(\texttt{spec}, \mathcal{S}),$$

where `res` is `true` if the specification is satisfied, and `false` otherwise.

## 7.4 Restructuring Polynomial Zonotopes

In this subsection, we describe the settings for triggering and implementing the `restructure` operation of polynomial zonotopes (see Sec. 2.2.1.5). As described in Sec. 4.2.6.1, it is advantageous to use a non-convex set representation such as polynomial zonotopes to represent the reachable sets of nonlinear systems. Since during reachability analysis the size of the independent part of the polynomial zonotope constantly grows, the accuracy can be significantly improved by shifting generators from the independent to the dependent part as done by the `restructure` operation described in [39, Sec. 2.5]. For this restructuring process, there exist some additional settings listed in Tab. 25.

## 7.5 Evaluating the Lagrange Remainder

One critical step in reachability analysis for nonlinear systems is the evaluation of the Lagrange remainder $\mathcal{L}$ (see (40) in Sec. 4.2.6.1) using range bounding (see Sec. 2.2.3). The evaluation of the Lagrange remainder is often the most time-consuming part of reachability analysis and if the computed bounds are not tight, the reachable set might "explode". Therefore, CORA provides several different options for evaluating the Lagrange remainder, which can be specified as fields of the struct `options.lagrangeRem` (see Tab. 27).

[29] `https://de.mathworks.com/help/symbolic/simplify-symbolic-expressions.html`

Table 25: Fields of the struct `options.polyZono` defining the settings for restructuring polynomial zonotopes (see [39, Sec. 2.5]).

| Setting | Description |
|---|---|
| – `.maxPolyZonoRatio` | upper bound $\mu_d$ for the volume ratio between the independent and dependent part of a polynomial zonotope (see [39, Line 18 in Alg. 1]). If the bound is exceeded, the polynomial zonotope is restructured. The default value is $\infty$ (no restructuring). |
| – `.maxDepGenOrder` | upper bound for the value $\frac{p}{n}$ after restructuring, where $p$ is the number of dependent polynomial zonotope factors (see Sec. 2.2.1.5) and $n$ is the system dimension. The default value is 20. |
| – `.restructureTechnique` | string specifying the method that is applied to restructure polynomial zonotopes. The string is composed of two parts `restructureTechnique` = `method` + `reductionTechnique`, where `method` represents the restructure strategy (see Tab. 26) and `reductionTechnique` represents the zonotope reduction technique (see Tab. 4). Note that the two parts are combined by camelCase. The default value is `'reduceGirard'`. |

Table 26: Strategies for restructuring polynomial zonotopes

| Strategy | Description |
|---|---|
| `reduce` | reduction of independent generators |
| `reduceFull` | reduction of independent generators to zonotope order 1 |
| `zonotope` | enclosure of polynomial zonotope with a zonotope |

Table 27: Fields of the struct `options.lagrangeRem` defining the settings for evaluating the Lagrange remainder during reachability analysis for nonlinear systems.

| Setting | Description |
|---|---|
| – `.simplify` | string specifying the method to simplify the symbolic equations in the Lagrange remainder. The available methods are `'none'` (no simplification), `'simplify'` (simplification using MATLABs `simplify` function[29]), `'collect'` (simplification using MATLABs `collect` function[30]), and `'optimize'` (simplifications using MATLABs code optimization for symbolic expressions[31]). The default value is `'none'`. |
| – `.tensorParallel` | flag with value 0 or 1 specifying whether parallel computing is used to evaluate the Lagrange remainder. The default value is 0 (no parallel computing). |
| – `.replacements` | function handle to a function $r(x, u)$ (nonlinear systems) or $r(x, u, p)$ (nonlinear parametric systems) that describes expressions that are replaced and precomputed in the Lagrange remainder equations in order to speed up the evaluation (optional). |
| – `.method` | range bounding method used for evaluating the Lagrange remainder. The available methods are `'interval'` (interval arithmetic, see Sec. 2.2.1.2), `'taylorModel'` (see Sec. 2.2.3.1), or `'zoo'` (see Sec. 2.2.3.3). The default value is `'interval'`. |
| – `.zooMethods` | cell array specifying the range bounding methods for class `zoo` (see Sec. 2.2.3.3). The available methods are `'interval'`, `'affine(int)'`, `'affine(bnb)'`, `'affine(bnbAdv)'`, `affine(linQuad)'`, `'taylm(int)'`, `'taylm(bnb)'`, `'taylm(bnbAdv)'`, and `'taylm(linQuad)'`. |
| – `.maxOrder` | maximum polynomial order for Taylor models (see Sec. 2.2.3.1). |
| – `.optMethod` | method used to calculate bounds of Taylor models (see Sec. 2.2.3.1). The available methods are `'int'`, `'bnb'`, `'bnbAdv'`, and `'linQuad'`. The default value is `'int'`. |
| – `.tolerance` | minimum absolute value for Taylor model coefficients (see Sec. 2.2.3.1). |
| – `.eps` | termination tolerance for bounding algorithm for Taylor models (see Sec. 2.2.3.1). |

## 7.6 Verified Global Optimization

For general nonlinear functions $f(x)$ it is often impossible to compute the global minimum or maximum. However, if the values for the variable $x$ are restricted to a certain domain, the approach from [111] can be applied to compute the minimum or maximum on the domain with a certain precision. In CORA, the approach from [111] is implemented in the method `globVerMinimization`: Given a nonlinear function $f : \mathbb{R}^n \to \mathbb{R}$ and an interval domain $\mathcal{D} \subset \mathbb{R}^n$, the method `globVerMinimization` computes the global minimum of $f(x)$ on $\mathcal{D}$ with precision $\epsilon$:

$$[\hat{f}^{opt}, \hat{x}^{opt}, \mathcal{D}^{opt}] = \texttt{globVerMinimization}(f, \mathcal{D}, \epsilon),$$

$$\text{with} \quad \hat{f}^{opt} \in f^{opt} \oplus [-\epsilon, \epsilon], \quad f^{opt} = \min_{x \in \mathcal{D}} f(x),$$
$$x^{opt} \in \mathcal{D}^{opt}, \quad x^{opt} = \arg\min_{x \in \mathcal{D}} f(x),$$

where $\hat{x}^{opt} \in \mathcal{D}^{opt}$ is the most likely position of the global minimum, the function $f(x)$ is provided as a MATLAB function handle, and the domain $\mathcal{D}$ is represented as an object of class `interval` (see Sec. 2.2.1.2).

Note that for computing the global maximum, one can just minimize the negated function $-f(x)$. To compute both, the minimum and the maximum, one can use the method `globVerBounds`.

To demonstrate verified global optimization in CORA, we consider the example of the Beale function (see [111, Sec. 6]), which has the global minimum $f^{opt} = 0$ at the point $x^{opt} = [3, 0.5]^T$:

```
% function f
f = @(x) (1.5 - x(1)*(1-x(2))).^2 + ...
         (2.25 - x(1)*(1-x(2)^2))^2 + ...
         (2.625 - x(1)*(1-x(2)^3))^2;

% domain D
D = interval([-4.5;-4.5],[4.5;4.5]);

% verified global optimization
[val,xOpt,domOpt] = globVerMinimization(f,D,1e-5);
```

```
Command Window:

val =
  -2.7163e-06

domOpt =

  [3.00037,3.00092]
  [0.49966,0.50011]
```

## 7.7 Kaucher Arithmetic

As described in Sec. 2.2.3, *interval arithmetic* [46] can be applied to compute an over-approximation for the range of values of a nonlinear function. In this section we consider *Kaucher arithmetic* [112], which returns intervals that are interpretable as inner-approximation of the range of values for nonlinear functions that can be rewritten or abstracted so that each variable appears at most once. Kaucher arithmetic is based on generalized intervals defined as

$$\mathcal{K} = [\underline{x}, \overline{x}], \quad \underline{x}, \overline{x} \in \mathbb{R}^n. \tag{54}$$

In contrast to intervals as introduced in Sec. 2.2.1.2, generalized intervals omit the constraint $\forall i = \{1, \ldots, n\} : \ \underline{x}_i \leq \overline{x}_i$. In CORA, generalized intervals are implemented by the class

[30] `https://de.mathworks.com/help/symbolic/collect.html`
[31] see setting `'Optimize'` in `https://de.mathworks.com/help/symbolic/matlabfunction.html`

`intKaucher`. An object of class `intKaucher` can be constructed as follows:

$$\mathcal{K} = \texttt{intKaucher}(\underline{x}, \overline{x}),$$

where $\underline{x}, \overline{x}$ are defined as in (54). We demonstrate Kaucher arithmetic using the example in [93, Example 1], which considers the nonlinear function $f(x) = x^2 - x$ and the domain $x \in [2, 3]$. Since the variable $x$ occurs twice in the function $f(x)$, Kaucher arithmetic cannot be applied directly. Therefore, we first compute an enclosure of the function $f(x)$ using the mean value theorem:

$$f^{abstract}(x) = f(2.5) + \left.\frac{\partial f(x)}{\partial x}\right|_{x \in [2,3]} (x - 2.5) = 3.75 + [3, 5](x - 2.5).$$

Since the variable $x$ occurs only once in the resulting function $f^{abstract}(x)$, we can now apply Kaucher arithmetic to compute an inner-approximation of the range of values for the function $f(x)$ on the domain $x \in [2, 3]$, which yields $\{f(x) \mid x \in [2, 3]\} \supseteq [2.25, 5.25]$. In CORA, this example can be implemented as follows:

```
% function f
f = @(x) x^2 - x;

% compute gradient
syms x;
df = gradient(f(x));
df = matlabFunction(df);

% compute bounds for gradient
I = interval(2,3);
c = center(I);
gr = df(I);

% compute inner-approximation of the range
x = intKaucher(3,2);
gr = intKaucher(infimum(gr),supremum(gr));

res = f(c) + gr*(x - c);
```

```
Command Window:

res =
[5.25000,2.25000]
```

## 7.8 Contractors

Contractor programming [46, Chapter 4] can be used to contract an interval domain of possible values with respect to one or multiple nonlinear constraints, which is useful for many applications. In CORA, contractor programming is implemented by the method `contract`: Given a constraint $f(x) = \mathbf{0}$ defined by a nonlinear function $f : \mathbb{R}^n \to \mathbb{R}^m$ and an interval domain $\mathcal{D} \subset \mathbb{R}^n$, the method `contract` returns a contracted interval

$$\widehat{\mathcal{D}} = \texttt{contract}(f, \mathcal{D}, \texttt{method}),$$
$$\widehat{\mathcal{D}} = \texttt{contract}(f, \mathcal{D}, \texttt{method}, \texttt{iter}),$$
$$\widehat{\mathcal{D}} = \texttt{contract}(f, \mathcal{D}, \texttt{method}, \texttt{splits}),$$

that satisfies

$$\left\{x \in \mathbb{R}^n \mid f(x) = \mathbf{0}, x \in \mathcal{D}\right\} \subseteq \widehat{\mathcal{D}},$$

where the function $f(x)$ is specified as a MATLAB function handle and $\mathcal{D}, \widehat{\mathcal{D}}$ are both represented as object of class `interval` (see Sec. 2.2.1.2). The additional input arguments are as follows:

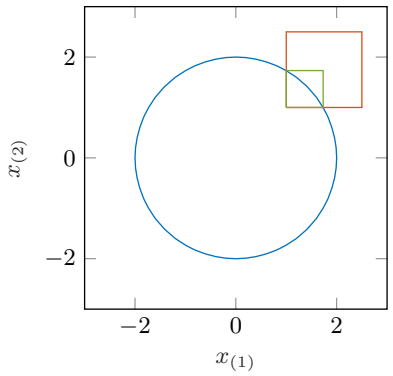
Table 28: Contractors implemented in CORA.

| Contractor | Description | Reference |
|---|---|---|
| `forwardBackward` | forward-backward traversion of the syntax tree | [46, Chapter 4.2.4] |
| `linearize` | parallel linearization of constraints | [46, Chapter 4.3.4] |
| `polyBox` | extremal functions of polynomial constraints | [113] |

- `method` string specifying the contractor that is used. The available contractors are listed in Tab. 28. If set to `'all'`, all available contractors are applied one after another.
- `iter` number of consequtive contractions. The default value is 1, so that the contractor is applied only once.
- `splits` number of iterative splits applied to the domain $\mathcal{D}$ in order to refine the result of the contraction. The default value is 0, so that no splitting is applied.

Let us demonstrate contractor programming in CORA by an example:

```
% function f(x)
f = @(x) x(1)^2 + x(2)^2 - 4;

% domain D
dom = interval([1;1],[2.5;2.5]);

% contraction
res = contract(f,dom,'forwardBackward');
```




## 7.9 Signal Temporal Logic

Signal temporal logic is a common formalism to represent complex specifications that describe the desired behavior of a system. In CORA, signal temporal logic formulas are represented by the class `stl`. An object of class `stl` can be constructed as follows:

$$\texttt{obj = stl(name,}n\texttt{)},$$

where `name` is a string specifying the name of the variable and $n$ is the dimension of the variable. The variables constructed with the constructor of the class `stl` correspond to the system states. These variables can be used to construct predicates with the operators `+`, `-`, `*`, `<`, `<=`, `>`, and `>=`. In addition, set containment $x \in \mathcal{S}$ can be realized with the function `in`$(x,\mathcal{S})$, where $\mathcal{S}$ is a continuous set (see Sec. 2.2). The predicates can then be used as inputs for the signal temporal logic operators in Tab. 29. Note that `stlInterval` objects, and thus the temporal operators, also support open and half-open intervals (see the example below). The resulting signal temporal logic formula can be used to construct a system specification (see Sec. 7.3) which is checked during reachability analysis, where CORA implements the approach in [59] to check if the reachable set satisfies temporal logic specifications. Moreover, the incremental verification algorithm for signal temporal logic from [7] is implemented in the method `modelChecking` of the class `reachSet`.

Let us demonstrate signal temporal logic in CORA by an example:

Table 29: Operators for signal temporal logic, where $\xi(t)$ is a trace and $\models$ denotes entailment.

| Operator | CORA | Definition |
|---|---|---|
| $\phi \wedge \psi$ | `p & q` | $\xi(t) \models \phi \wedge \xi(t) \models \psi$ |
| $\phi \vee \psi$ | `p \| q` | $\xi(t) \models \phi \vee \xi(t) \models \psi$ |
| $\neg \phi$ | `~p` | $\xi(t) \models \neg\phi$ |
| $\phi \Rightarrow \psi$ | `implies(p,q)` | $\xi(t) \models \phi \Rightarrow \xi(t) \models \psi$ |
| $\mathcal{X}_a \, \phi$ | `next(p,a)` | $\xi(t+a) \models \phi$ |
| $\mathcal{F}_{[a,b]} \, \phi$ | `finally(p,stlInterval(a,b))` | $\exists t \in [a,b] : \xi(t) \models \phi$ |
| $\mathcal{G}_{[a,b]} \, \phi$ | `globally(p,stlInterval(a,b))` | $\forall t \in [a,b] : \xi(t) \models \phi$ |
| $\phi \mathcal{U}_{[a,b]} \, \psi$ | `until(p,q,stlInterval(a,b))` | $\exists t \in [a,b] : \xi(t) \models \psi \wedge \forall t' \in [0,t) : \xi(t') \models \phi$ |
| $\phi \mathcal{R}_{[a,b]} \, \psi$ | `release(p,q,stlInterval(a,b))` | $\forall t \in [a,b] : \xi(t) \models \psi \vee \exists t' \in [0,t) : \xi(t') \models \phi$ |

```
% create variable
x = stl('x',2);

% signal temporal logic formula
eq = until(x(1) < 3,x(2) > 5, ...
    stlInterval(1,3,true,false))
```

```
Command Window:

eq =

(x1 < 3 U[1,3) x2 > 5)
```

## 7.10 Conversion of CommonRoad Models

CommonRoad[32] is a collection of composable benchmarks for motion planning on roads. The syntax for loading a CommonRoad file with the function `commonroad2cora` is as follows:

$$[\texttt{statObs}, \texttt{dynObs}, \texttt{x0}, \texttt{goalSet}, \texttt{lanelets}] = \texttt{commonroad2cora}(\texttt{filename}),$$

where `filename` is a string with the file name of the CommonRoad file that should be loaded, and the output arguments are defined as:

- `statObs` MATLAB cell-array storing the static obstacles for the planning problem as objects of class `polygon` (wrapper class for MATLAB class `polyshape`).
- `dynObs` MATLAB cell-array storing the dynamic obstacles for the planning problem as objects of class `polygon` (wrapper class for MATLAB class `polyshape`). In addition, the corresponding time interval for each obstacle is stored.
- `x0` struct with fields `.x`, `.y`, `.time`, `.velocity` and `.orientation` storing the initial state for the planning problem.
- `goalSet` struct with fields `.set`, `.time`, `.velocity` and `.orientation` storing the goal set for the planning problem.
- `lanelets` MATLAB cell-array storing the lanelets for the traffic scenario as objects of class `polygon` (wrapper class for MATLAB class `polyshape`).

[32] `commonroad.in.tum.de`

# 8 Loading Simulink and SpaceEx Models

Since CORA 2018 it is possible load SpaceEx models. This not only has the advantage that one can use the SpaceEx model editor to create models for CORA (see Sec. 8.1.2), but also makes it possible to indirectly load Simulink models through the SL2SX converter [114,115] (see Sec. 8.1.1). Since CORA 2020 it is furthermore possible to export CORA models as SpaceEx models (see Sec. 4.1.11), which closes the loop between the two formats. We also plan to make the conversion to CORA available within HYST in the future [116]. We first present how to create SpaceEx models and then how one can convert them to CORA models.

## 8.1 Creating SpaceEx Models

We present two techniques to create SpaceEx models: a) converting Simulink models to SpaceEx models and b) creating models using the SpaceEx model editor.

### 8.1.1 Converting Simulink Models to SpaceEx Models

The SL2SX converter generates SpaceEx models from Simulink models and can be downloaded from github.com/nikos-kekatos/SL2SX.

After downloading the SL2SX converter or cloning it using the command

`git clone https://github.com/nikos-kekatos/SL2SX.git`,

one can run the tool using the Java Runtime Environment, which is pre-installed on most systems. You can check whether it is pre-installed by typing `java -version` in your terminal. To run the tool, type `java -jar SL2SX.jar`. One can also run the converter directly in the MATLAB command window by typing

```
system(sprintf('java -jar path_to_converter/SL2SX_terminal.jar %s', ...
'path_to_model/model_name.xml'))
```

after adding the files of the converter to the MATLAB path, where the placeholders `path_to_converter` and `path_to_model` represent the corresponding file paths.

To use the converter, you have to save your Simulink model in XML format by typing in the MATLAB command window:

```
load_system('model_name')
save_system('model_name.slx','model_name.xml','ExportToXML',true)
```

When the model is saved as `*.mdl` instead of `*.slx`, please replace `'model_name.slx'` by `'model_name.mdl'` above. A screenshot of an example to save a model in XML format together with the corresponding Simulink model of a DC motor is shown in Fig. 22.

Please note that the SL2SX converter cannot convert any Simulink model to SpaceEx. A detailed description of limitations can be found in [114, 115].

### 8.1.2 SpaceEx Model Editor

To create SpaceEx models in an editor, one can use the SpaceEx model editor downloadable from spaceex.imag.fr/download-6.

To use the editor, save the file (e.g., spaceexMOE.0.9.4.jar) and open a terminal. To execute the model editor, type `java -jar filename.jar` and in the case of the example file, type `java -jar spaceexMOE.0.9.4.jar`. If it does not work, you might want to check if you have java installed: type `java -version` in your terminal.

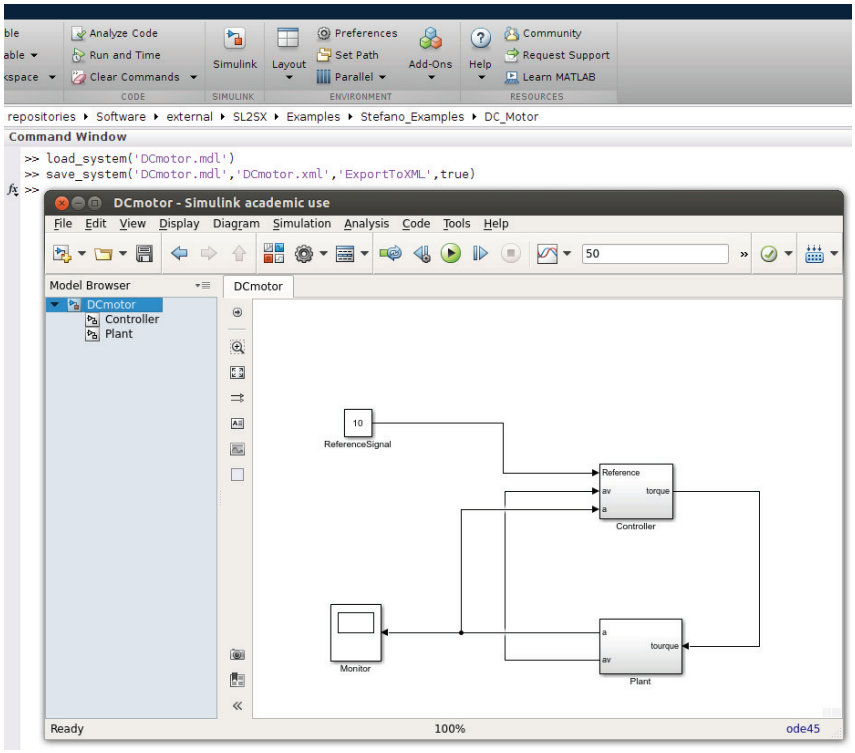



Figure 22: Screenshot of MATLAB/Simulink showing how to save Simulink models in XML format.

A screenshot of the model editor can be found in Fig. 23. Further information on the SpaceEx modeling language can be found in [65] and further documents can be downloaded from: spaceex.imag.fr/documentation/user-documentation.

Examples of SpaceEx models can be loaded in CORA from `/models/SpaceEx`.

## 8.2 Converting SpaceEx Models

To load SpaceEx models (stored as XML files) into CORA, one only has to execute a simple command:

```
spaceex2cora('model.xml');
```

This command creates a CORA model in `/models/SpaceExConverted` under a folder with the identical name as the SpaceEx model. If the SpaceEx model contains nonlinear differential equations, additional dynamics files are stored in the same folder. Below, we present as an example the converted model of the bouncing ball from SpaceEx:

```
function HA = bball(~)

%% Generated on 27-Aug-2022

%---------------Automaton created from Component 'system'------------------

%% Interface Specification:
%   This section clarifies the meaning of state, input & output dimensions
%   by showing their mapping to SpaceEx variable names.

% Component 1 (system.ball):
%  state x := [x; v]
%  input u := [uDummy]

%-------------------------Component system.ball-----------------------------

%-----------------------------State always----------------------------------

%% equation:
%   x' == v & v' == -g
dynA = ...
[0,1;0,0];
dynB = ...
[0;0];
dync = ...
[0;-9.81];
dynamics = linearSys(dynA, dynB, dync);

%% equation:
%   x >= 0
A = ...
[-1,0];
b = ...
[0];
polyOpt = struct('A', A, 'b', b);
inv = polytope(polyOpt);

trans = {};
%% equation:
%   v' := -c*v
resetA = ...
[1,0;0,-0.75];
resetB = ...
[1,0;0,-0.75];
resetc = ...
[0;0];
reset = struct('A', resetA, 'B', resetB, 'c', resetc);

%% equation:
%   x <= eps & v < 0
c = [-1;0];
d = 0;C = ...
[0,1];
D = [0];

guard = polytope(C,D,c,d);

trans(1) = transition(guard, reset, 1);

loc(1) = location('S1', inv, trans, dynamics);

HA = hybridAutomaton(loc);
```

```
end
```

At the beginning of each automatically-created model, we list the states and inputs so that the created models can be interpreted more easily using the variable names from the SpaceEx model. These variable names are later replaced by the state vector $x$ and the input vector $u$ to make use of matrix multiplications in MATLAB for improved efficiency. Next, the dynamic equations, guard sets, invariants, transitions, and locations are created (the semantics of these components is explained in Sec. 4.3).

A hand-written version of the bouncing ball example can be found in Sec. 10.4.1 for comparison.

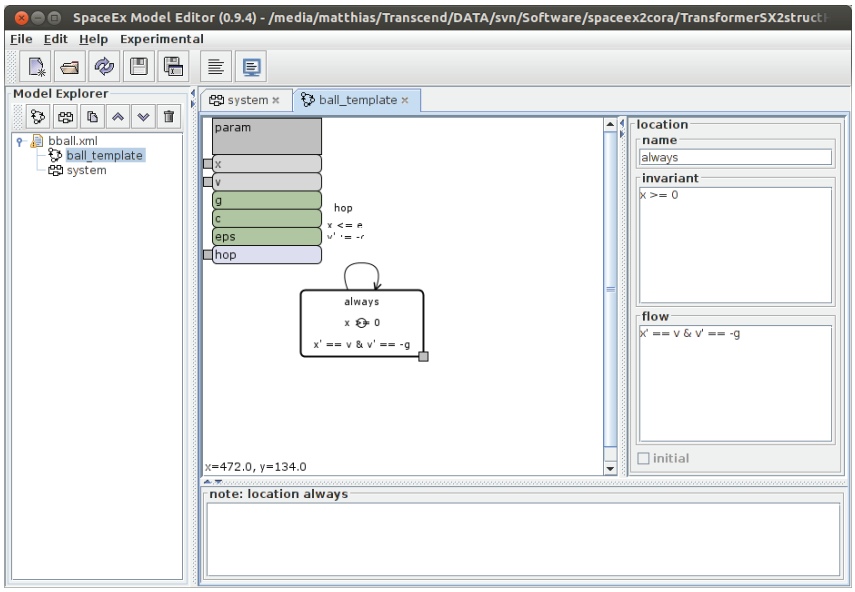



Figure 23: Screenshot of the SpaceEx model editor showing the bouncing ball example.

### Remarks

1. The converter makes heavy use of operations of strings, which have been modified since MATLAB 2017a. We have developed the converter using MATLAB 2017b. It is thus recommended to update to the latest MATLAB version to use the converter. It cannot be used if you have a version older than 2017a.

2. It is not yet possible to convert all possible models that can be modeled in SpaceEx. This is mostly due to unfinished development of the converter. Some cases, however, are due to the less strict hybrid automaton definition used by SpaceEx, which allows for models that currently cannot be represented in CORA. Hybrid models (see Sec. 4.3) that do not violate the following restrictions can be converted:

   - **Uncertain parameters**: CORA supports models with varying parameters, but our converter cannot produce such models yet. Parameters must be fixed in the SpaceEx model or will be treated as time-varying inputs. This may result in nonlinear differential equations even when the system is linear time-varying.

   - **Reset Functions**: Resets can be linear: $x' = Ax + Bu + c$, where $x'$ is the state after the reset, $A \in \mathbb{R}^{n \times n}$, $x \in \mathbb{R}^n$ is the state before the reset, and $B \in \mathbb{R}^{n \times m}$, $d \in \mathbb{R}^n$. They may also be nonlinear functions $x' = f(x, u)$. Resets violating this restriction are ignored and trigger a warning.

- **Local Variables**: Our parser can currently not detect local variables that are defined in bound components but not in the root component (detailed definitions of local variables, bound components, and root components can be found in [117]). Therefore all relevant variables are required to be non-local in all components.
- **Labels**: Synchronization labels (variables of type `label`) are also supported.

3. SX2CORA does not keep all inputs of the SpaceEx Model, if they have no effect on the generated model (i.e., inputs/uncertain parameters that were only used in invariants/-guards/resets).
4. Variable names **i j I J** are renamed to **ii jj II JJ**, since the MATLAB Symbolic Toolbox would interpret them as the imaginary number. Variables such as **ii III JJJJ** are also lengthened by a letter to preserve name uniqueness.

### Optional arguments

To better control the conversion, one can use additional arguments:

```
spaceex2cora('model.xml','rootID','outputName','outputDir','cfgFile');
```

The optional arguments are:

- `'rootID'` – ID of SpaceEx component to be used as root component (specified as a string).
- `'outputName'` – name of the generated CORA model (specified as a string).
- `'outputDir'` – path to the desired output directory where all generated files are stored (specified as a string).
- `'cfgFile'` – path to the file containing the SpaceEx configuration (specified as a string).

The implementation of the SX2CORA converter is described in detail in Appendix D.

# 9 Graphical User Interface

Since the 2021 release the CORA toolbox includes a graphical user interface (GUI). This GUI provides access to CORA's main functionality, even for users without any knowledge about programming in MATLAB. We recommend the usage of the GUI especially for CORA beginners since it allows to select algorithm settings conveniently using the respective drop-down menus. Moreover, the GUI contains info buttons that display detailed descriptions for all algorithm settings.

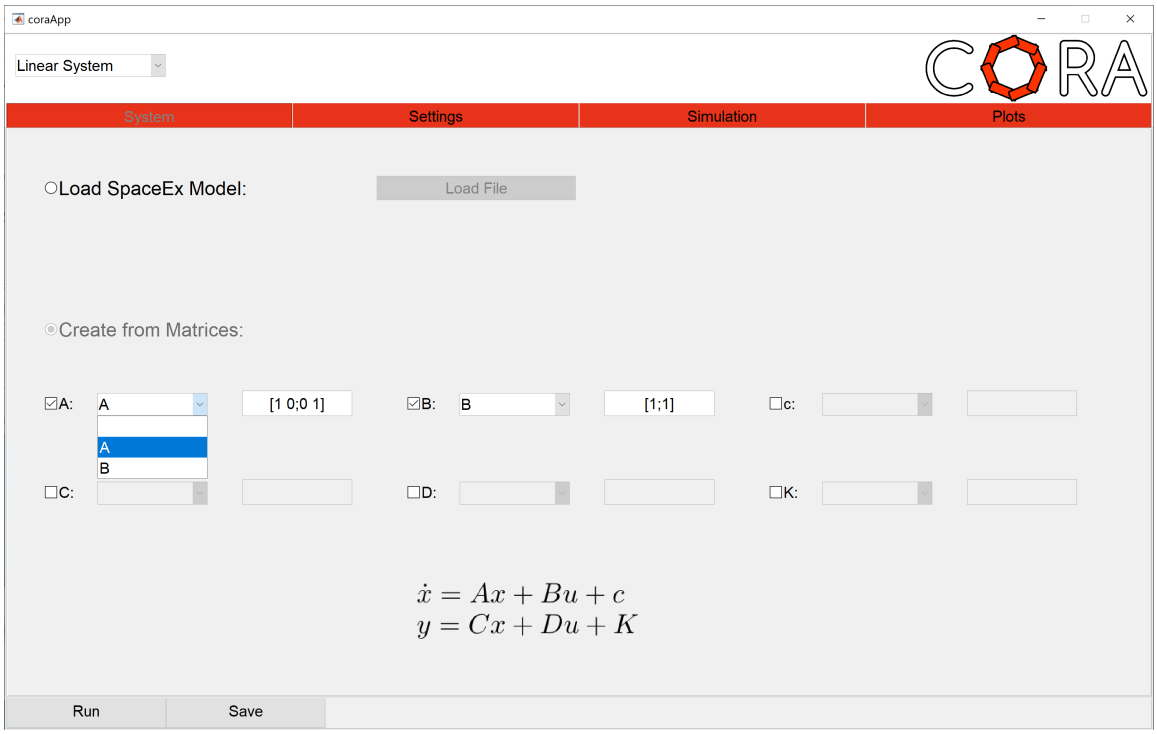



Figure 24: Screenshot of the graphical user interface for CORA.

The GUI can be started by running

```
>> coraApp
```

form the MATLAB command window. In particular, the GUI can be used to compute reachable sets, simulate trajectories, and visualize the corresponding results for linear continuous systems (see Sec. 4.2.1), nonlinear continuous systems (see Sec. 4.2.6), and hybrid automata (see Sec. 4.3.1). A screenshot from the GUI is shown in Fig. 24. To specify parameters, such as the system matrix $A$ for linear systems in Fig. 24, the GUI provides two options: One can either select variables from the MATLAB workspace using the drop-down menu, or specify parameters manually in the text field on the right hand side of the drop-down menu. If your workspace variables do not appear in the drop-down menu, please press the refresh workspace variables button at the top right. After one has specified all parameters and settings, the GUI offers two functionalities: By clicking on the run-button on the bottom left (see Fig. 24), the GUI generates figures that visualize the corresponding results. On the other hand, by clicking on the save-button the GUI generates a MATLAB script containing the corresponding CORA code.

# 10 Examples

This section presents a variety of examples that have been published in different papers. For each example, we provide a reference to the paper so that the details of the system can be studied there. The focus of this manual is on how the examples in the papers can be realized using CORA—this, of course, is not shown in scientific papers due to space restrictions.

## 10.1 Set Representations

We first provide examples for set-based computation using the different set representations in Sec. 2.

### 10.1.1 Zonotopes

The following MATLAB code demonstrates how to perform set-based computations on zonotopes (see Sec. 2.2.1.1):

```
Z1 = zonotope([1 1 1; 1 -1 1]); % create zonotope Z1
Z2 = zonotope([-1 1 0; 1 0 1]); % create zonotope Z2
A = [0.5 1; 1 0.5]; % numerical matrix A

Z3 = Z1 + Z2; % Minkowski addition
Z4 = A*Z3; % linear map

figure; hold on
plot(Z1,[1 2],'b'); % plot Z1 in blue
plot(Z2,[1 2],'g'); % plot Z2 in green
plot(Z3,[1 2],'r'); % plot Z3 in red
plot(Z4,[1 2],'k'); % plot Z4 in black

P = polytope(Z4) % convert to and display halfspace representation
I = interval(Z4) % convert to and display interval

figure; hold on
plot(Z4); % plot Z4
plot(I,[1 2],'g'); % plot intervalhull in green
```

This produces the workspace output

```
Normalized, minimal representation polytope in R^2
H: [8x2 double]
K: [8x1 double]
normal: 1
minrep: 1
xCheb: [2x1 double]
RCheb: 1.4142

[ 0.70711     0.70711]      [  6.364]
[ 0.70711    -0.70711]      [ 2.1213]
[ 0.89443    -0.44721]      [ 3.3541]
[ 0.44721    -0.89443]      [ 2.0125]
[-0.70711    -0.70711] x <= [ 2.1213]
[-0.70711     0.70711]      [0.70711]
[-0.89443     0.44721]      [0.67082]
[-0.44721     0.89443]      [ 2.0125]
```

```
Intervals:
[-1.5,5.5]
[-2.5,4.5]
```

The plots generated in lines 9-12 are shown in Fig. 25 and the ones generated in lines 18-19 are shown in Fig. 26.

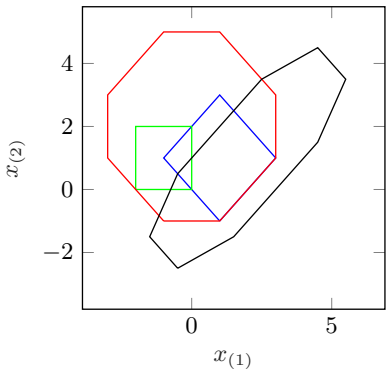



Figure 25: Zonotopes generated in lines 9-12 of the zonotope example in Sec. 10.1.1.

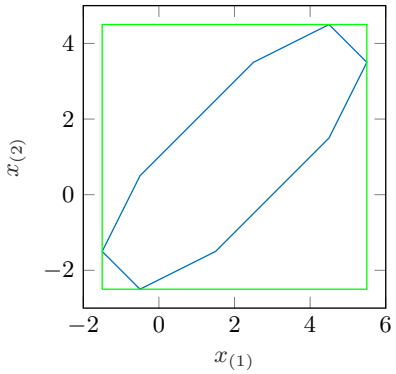



Figure 26: Sets generated in lines 18-19 of the zonotope example in Sec. 10.1.1.

### 10.1.2 Intervals

The following MATLAB code demonstrates how to perform set-based computations on intervals (see Sec. 2.2.1.2):

```
1  I1 = interval([0; -1], [3; 1]); % create interval I1
2  I2 = interval([-1; -1.5], [1; -0.5]); % create interval I2
3  Z1 = zonotope([1 1 1; 1 -1 1]); % create zonotope Z1
4
5  r = rad(I1) % obtain and display radius of I1
6  is_intersecting = isIntersecting(I1, Z1) % Z1 intersecting I1?
7  I3 = I1 & I2; % computes the intersection of I1 and I2
8  c = center(I3) % returns and displays the center of I3
9
10 figure; hold on
11 plot(I1); % plot I1
12 plot(I2); % plot I2
13 plot(Z1,[1 2],'g'); % plot Z1
14 plot(I3,[1 2],'FaceColor',[.6 .6 .6]); % plot I3
```

This produces the workspace output

```
r =

    1.5000
    1.0000


is_intersecting =

    1
```

```
c =

    0.5000
   -0.7500
```

The plot generated in lines 11-14 is shown in Fig. 27.

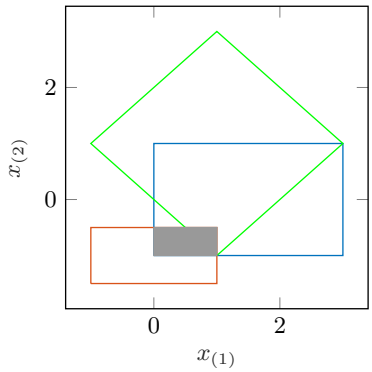



Figure 27: Sets generated in lines 11-14 of the interval example in Sec. 10.1.2.

### 10.1.3 Ellipsoids

The following MATLAB code demonstrates how to perform set-based computations on ellipsoids (see Sec. 2.2.1.3):

```
1  E1 = ellipsoid(diag([1/2,2])) % create ellipsoid E1 and display it
2  A = diag([2,0.5]);
3
4  E2 = A*E1 + 0.5; % linear Map + Minkowski addition
5  E3 = E1 + E2; % Minkowski addition
6  E4 = E1 & E2; % intersection
7
8  disp(['E1 in E2?: ',num2str(E2.contains(E1))]);
9  disp(['E1 in E3?: ',num2str(E3.contains(E1))]);
10
11 figure; hold on
12 plot(E1,[1,2],'b'); % plot E1 in blue
13 plot(E2,[1,2],'g'); % plot E2 in green
14 plot(E3,[1,2],'r'); % plot E3 in red
15 plot(E4,[1,2],'k'); % plot E4 in black
16
17 E5 = ellipsoid([0.8,-0.6; -0.6,0.8],[1; -4]); % create ellipsoid E5
18 Zo_box = zonotope(E5); % overapproximate E5 by a parallelotope
19 Zu_norm = zonotope(E5,10,'outer:norm'); % overapproximate E5 using zonotope norm
20
21 figure; hold on
22 plot(E5); % plot E5
23 plot(Zo_box,[1,2],'r'); % plot overapproximative zonotope Zo_box
24 plot(Zu_norm,[1,2],'m');% plot overapproximative zonotope Zu_norm
```

This produces the workspace output

```
E1 =

ellipsoid:
```

```
- dimension: 2

q:
0
0

Q:
0.5000          0
0          2.0000

dimension:
2

degenerate:
0

E1 in E2?: 0
E1 in E3?: 1
```

The plots generated in lines 12-15 are shown in Fig. 28 and the ones generated in lines 22-24 are shown in Fig. 29.

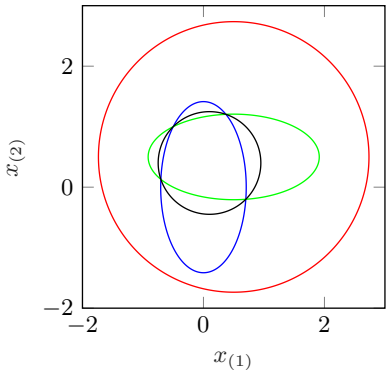



Figure 28: Ellipsoids generated in lines 12-15 of the ellipsoid example in Sec. 10.1.3.

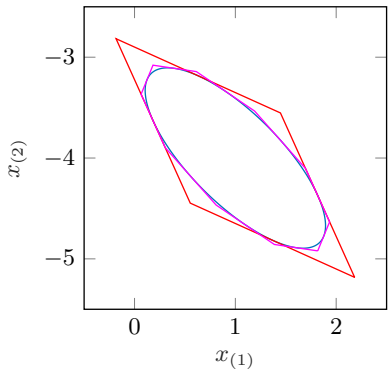



Figure 29: Sets generated in lines 22-24 of the ellipsoid example in Sec. 10.1.3.

### 10.1.4 Polytopes

The following MATLAB code demonstrates how to perform set-based computations on polytopes (see Sec. 2.2.1.4):

```
Z1 = zonotope([1 1 1; 1 -1 1]); % create zonotope Z1
Z2 = zonotope([-1 1 0; 1 0 1]); % create zonotope Z2

P1 = polytope(Z1); % convert zonotope Z1 to halfspace representation
P2 = polytope(Z2); % convert zonotope Z2 to halfspace representation

P3 = P1 + P2 % perform Minkowski addition and display result
P4 = P1 & P2; % compute intersection of P1 and P2

V = vertices(P4) % obtain and display vertices of P4

figure; hold on
plot(P1); % plot P1
plot(P2); % plot P2
plot(P3,[1 2],'g'); % plot P3
plot(P4,[1 2],'FaceColor',[.6 .6 .6]); % plot P4
```

This produces the workspace output

```
P3 =

polytope:
- dimension: 2

Vertex representation: (not computed)
Inequality constraints (A*x <= b):

A =

         0    1.0000
    0.7071    0.7071
    1.0000         0
         0   -1.0000
    0.7071   -0.7071
   -0.7071    0.7071
   -1.0000         0
   -0.7071   -0.7071


b =

    5.0000
    4.2426
    3.0000
    1.0000
    1.4142
    4.2426
    3.0000
    1.4142
```

```
Equality constraints (Ae*x = be): (none)

Bounded?                            true
Empty set?                          Unknown
Full-dimensional set?               Unknown
Minimal halfspace representation? Unknown
Minimal vertex representation?    Unknown

V:
         0   -1.0000         0
         0    1.0000    2.0000
```

The plot generated in lines 13-16 is shown in Fig. 30.

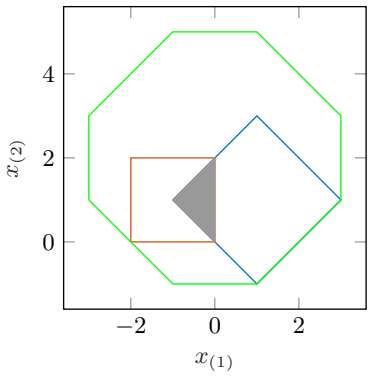



Figure 30: Sets generated in lines 13-16 of the polytope example in Sec. 10.1.4.

### 10.1.5 Polynomial Zonotopes

The following MATLAB code demonstrates how to perform set-based computations on polynomial zonotopes (see Sec. 2.2.1.5):

```
1  % construct zonotope
2  c = [1;0];
3  G = [1 1;1 0];
4  Z = zonotope(c,G);
5
6  % compute over-approximation of the quadratic map
7  Q{1} = [0.5 0.5; 0 -0.5];
8  Q{2} = [-1 0; 1 1];
9
10 resZono = quadMap(Z,Q);
11
12 % convert zonotope to polynomial zonotope
13 pZ = polyZonotope(Z);
14
15 % compute the exact quadratic map
16 resPolyZono = quadMap(pZ,Q);
17
18 % visualization
19 figure; hold on;
20 plot(resZono,[1,2],'r');
21 plot(resPolyZono,[1,2],'b');
```

The plot generated in lines 19-21 is shown in Fig. 31.

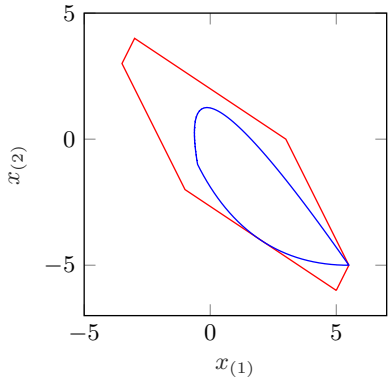



Figure 31: Quadratic map calculated with zonotopes (red) and polynomial zonotopes (blue).

### 10.1.6 Constrained Polynomial Zonotopes

The following MATLAB code demonstrates how to perform set-based computations on constrained polynomial zonotopes (see Sec. 2.2.1.6):

```
% construct zonotope
Z = zonotope([0;0],[1 1;0 1]);

% construct ellipsoid
E = ellipsoid([2 1;1 2],[1;1]);

% convert sets to constrained polynomial zonotopes
cPZ1 = conPolyZono(Z);
cPZ2 = conPolyZono(E);

% compute the Minkowski sum
resSum = cPZ1 + cPZ2;

% compute the intersection
resAnd = cPZ1 & cPZ2;

% compute the union
resOR = cPZ1 | cPZ2;

% construct conPolyZono object
c = [0;0];
G = [1 0 1 -1;0 1 1 1];
E = [1 0 1 2;0 1 1 0;0 0 1 1];
A = [1 -0.5 0.5];
b = 0.5;
R = [0 1 2;1 0 0;0 1 0];

cPZ = conPolyZono(c,G,E,A,b,R);

% compute quadratic map
Q{1} = [0.5 0.5; 0 -0.5];
Q{2} = [-1 0; 1 1];

res = quadMap(cPZ,Q);

% visualization
figure; hold on
plot(cPZ,[1,2],'b');

figure; hold on
plot(res,[1,2],'r','Splits',25);
```

The plot generated in lines 37-41 is shown in Fig. 32 and Fig. 33.

### 10.1.7 Capsules

The following MATLAB code demonstrates how to perform set-based computations on capsules (see Sec. 2.2.1.7):

```
% construct a capsule
c = [1;2];
g = [2;1];
r = 1;
```

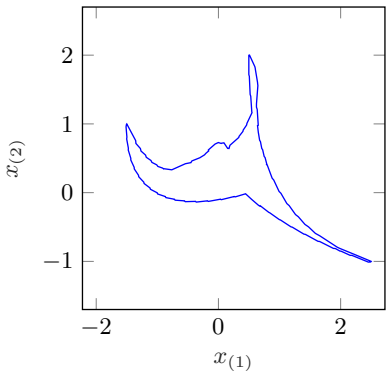



Figure 32: Constrained polynomial zonotope generated in lines 21-28 of the constrained polynomial zonotope example in Sec. 10.1.6

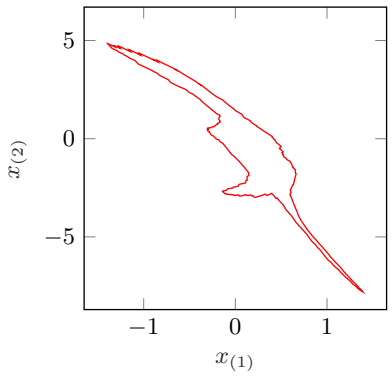



Figure 33: Quadratic map computed in lines 31-34 of the constrained polynomial zonotope example in Sec. 10.1.6.

```
5
6  C1 = capsule(c,g,r)
7
8  % linear map of a capsule
9  A = [0.5 0.2; -0.1 0.4];
10 C2 = A * C1;
11
12 % shift the center of a capsule
13 s = [0;1];
14 C3 = C2 + s;
15
16 % check capsule-in-capsule containment
17 res1 = contains(C1,C2);
18 res2 = contains(C1,C3);
19
20 disp(['C2 in C1?: ',num2str(res1)]);
21 disp(['C3 in C1?: ',num2str(res2)]);
22
23
24 % visualization
25 figure; hold on
26 plot(C1,[1,2],'r');
27 plot(C2,[1,2],'g');
28 plot(C3,[1,2],'b');
```

This produces the workspace output

```
id: 0
dimension: 2
center:
     1
     2

generator:
     2
     1

radius:
```

```
     1

C2 in C1?: 0
C3 in C1?: 1
```

The plot generated in lines 25-28 is shown in Fig. 34.

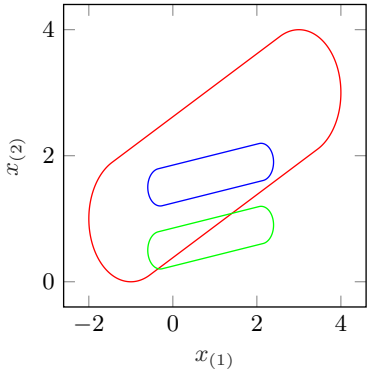



Figure 34: Capsules generated in lines 6, 10, and 14 of the capsule example in Sec. 10.1.7.

### 10.1.8 Zonotope Bundles

The following MATLAB code demonstrates how to perform set-based computations on zonotope bundles (see Sec. 2.2.1.8):

```
1 Z{1} = zonotope([1 1 1; 1 -1 1]); % create zonotope Z1;
2 Z{2} = zonotope([-1 1 0; 1 0 1]); % create zonotope Z2;
3 Zb = zonoBundle(Z); % instantiate zonotope bundle from Z1, Z2
4 vol = volume(Zb) % compute and display volume of zonotope bundle
5
6 figure; hold on
7 plot(Z{1}); % plot Z1
8 plot(Z{2}); % plot Z2
9 plot(Zb,[1 2],'FaceColor',[.675 .675 .675]); % plot Zb in gray
```

This produces the workspace output

```
vol =

    1.0000
```

The plot generated in lines 7-9 is shown in Fig. 35.

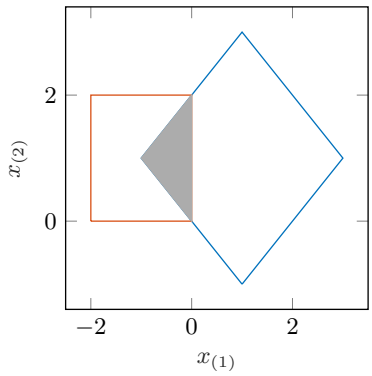



Figure 35: Sets generated in lines 7-9 of the zonotope bundle example in Sec. 10.1.8.

### 10.1.9 Constrained Zonotopes

The following MATLAB code demonstrates how to perform set-based computations on constrained zonotopes (see Sec. 2.2.1.9):

```
1 Z = [0 1 0 1; 0 1 2 -1]; % zonotope (center + generators)
2 A = [-2 1 -1]; % constraints (matrix A)
2 b = 2; % constraints (vector b)
3
4 cZ = conZonotope(Z,A,b) % construct conZonotope object
6
7 plotZono(cZ,[1,2]) % visualize conZonotope object + linear zonotope
```

This produces the workspace output

```
id: 0
dimension: 2
c:
     0
     0
```

```
g_i:
     1     0     1
     1     2    -1

A:
    -2     1    -1

b:
     2
```

The plot generated in line 9 is shown in Fig. 36. Fig. 37 displays a visualization of the constraints for the `conZonotope` object that is shown in Fig. 36.

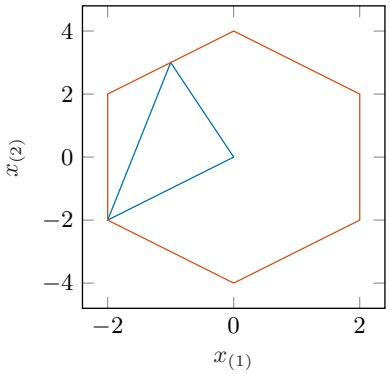



Figure 36: Zonotope (red) and the corresponding constrained zonotope (blue) generated in the constrained zonotope example in Sec. 10.1.9

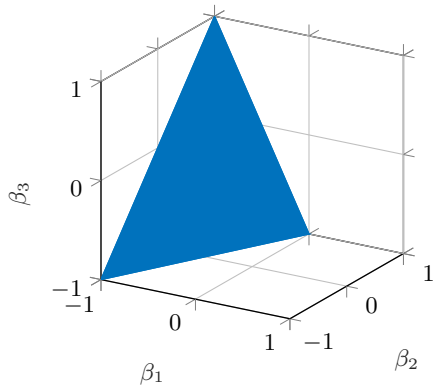



Figure 37: Visualization of the constraints for the `conZonotope` object generated in the constrained zonotope example in Sec. 10.1.9.

### 10.1.10 Spectrahedral Shadows

The following MATLAB code demonstrates how to perform set-based computations on spectrahedral shadows (see Sec. 2.2.1.10):

```
% Any convex set representation implemented in CORA can be represented as a
% spectrahedral shadow. For instance, ellipsoids, capsules, polytopes, and
% zonotopes can be recast as spectrahedral shadows:
E = ellipsoid([2 1; 1 2], [3;1]);
SpS_ellipsoid = spectraShadow(E);

C = capsule([-1;-3], [1;-2], 1);
SpS_capsule = spectraShadow(C);

P = polytope([1 0 -1 0 1; 0 1 0 -1 1]', [3; 2; 3; 2; 1]);
SpS_polytope = spectraShadow(P);

Z = zonotope([3;2], [0.5 1 0; 0 0.5 2.5]);
SpS_zonotope = spectraShadow(Z);

% perform Minkowski addition and display result
SpS_addition = SpS_polytope + SpS_ellipsoid;
% compute convex hull of C and Z
SpS_convHull = convHull(SpS_capsule, SpS_zonotope);

figure; hold on;
% One can now plot the convex hull of C and Z
plot(SpS_convHull,[1 2],'FaceColor',colorblind('gray'));
plot(C,[1 2],'Color',colorblind('r')); % plot C
plot(Z,[1 2],'Color',colorblind('b')); % plot Z
```

The plot generated in lines 21-25 is shown in Fig. 38.

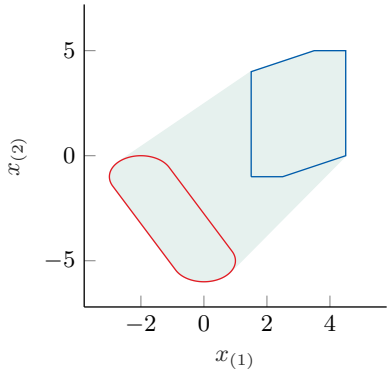



Figure 38: Sets generated in lines 21-25 of the spectrahedral shadow example in Sec. 10.1.10.

### 10.1.11 Probabilistic Zonotopes

The following MATLAB code demonstrates how to compute with probabilistic zonotopes (see Sec. 2.2.1.11):

```
Z1=[10; 0]; % uncertain center
Z2=[0.6 1.2  ; 0.6 -1.2]; % generators with normally distributed factors
pZ=probZonotope(Z1,Z2); % probabilistic zonotope

M=[-1 -1;1 -1]*0.2; % mapping matrix
```

```
6  pZencl = enclose(pZ,M); % probabilistic enclosure of pZ and M*pZ
7
8  figure % initialize figure
9  hold on
10 camlight headlight
11
12 plot(pZ,[1 2],'FaceColor',[0.2 0.2 0.2],...
13     'EdgeColor','none', 'FaceLighting','phong'); % plot pZ
14
15 plot(expm(M)*pZ,[1,2],'FaceColor',[0.5 0.5 0.5],...
16     'EdgeColor','none', 'FaceLighting','phong'); % plot expm(M)*pZ
17
18 plot(pZencl,[1,2],'k','FaceAlpha',0) % plot enclosure
19
20 campos([-3,-51,1]); % set camera position
21 drawnow; % draw 3D view
```

The plot generated in lines 8-21 is shown in Fig. 39.

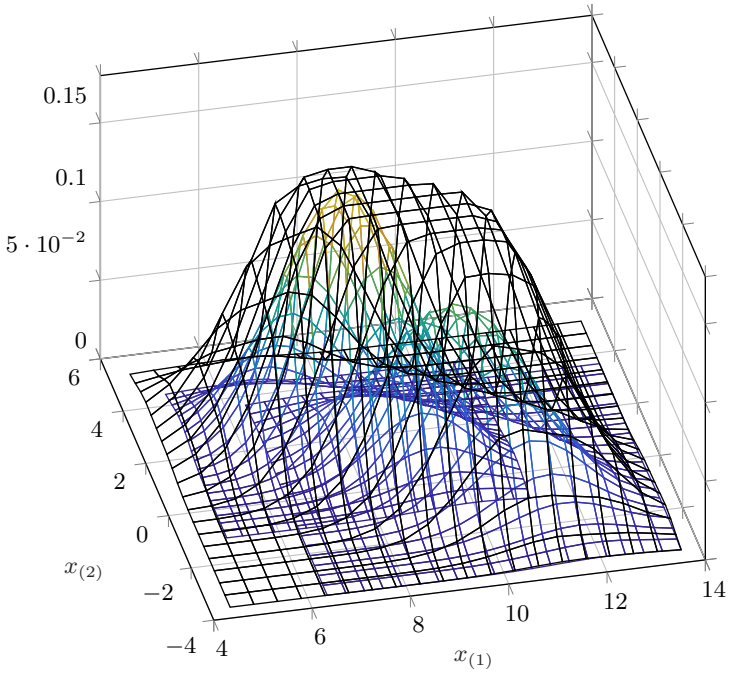



Figure 39: Sets generated in lines 10-15 of the probabilistic zonotope example in Sec. 10.1.11.

### 10.1.12 Level Sets

The following MATLAB code demonstrates how to compute with level sets (see Sec. 2.2.2.3):

```
% construct level sets
syms x y
eq = sin(x) + y;

ls1 = levelSet(eq,[x;y],'==');
ls2 = levelSet(eq,[x;y],'<=');

% visualize the level sets
subplot(1,2,1)
xlim([-1.5,1.5]);
ylim([-1,1]);
plot(ls1,[1,2],'b');

subplot(1,2,2)
xlim([-1.5,1.5]);
ylim([-1,1]);
plot(ls2,[1,2],'Color',[0.9451 0.5529 0.5686]);
```

The generated plot is shown in Fig. 40.

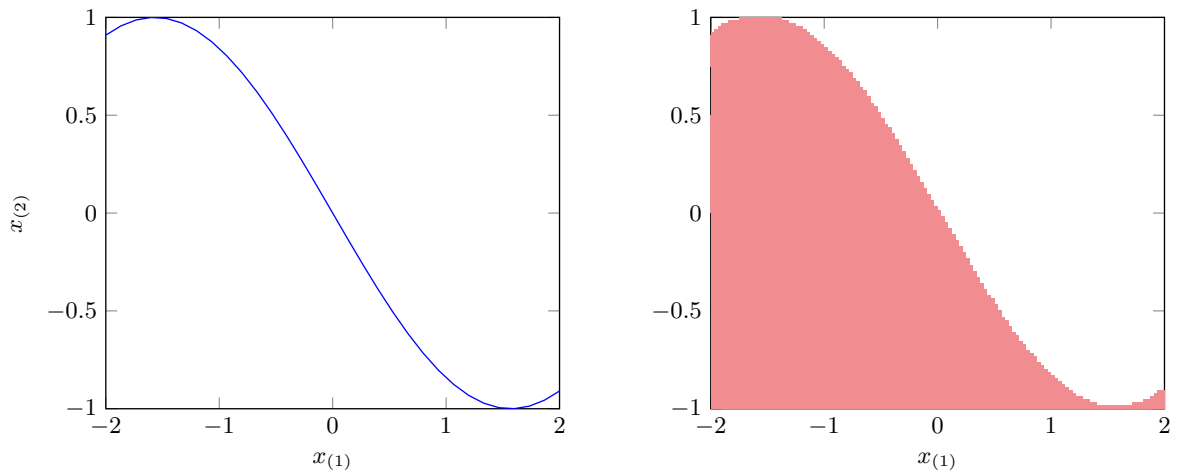



Figure 40: Level sets from the example in Sec. 10.1.12 defined as in (18) (left) and as in (20) (rigth).

### 10.1.13 Taylor Models

The following MATLAB code demonstrates how to compute with Taylor models (see Sec. 2.2.3.1):

```
a1 = interval(-1, 2); % generate a scalar interval [-1,2]
a2 = interval(2, 3); % generate a scalar interval [2,3]
a3 = interval(-6, -4); % generate a scalar interval [-6,4]
a4 = interval(4, 6); % generate a scalar interval [4,6]

b1 = taylm(a1, 6); % Taylor model with maximum order of 6 and name a1
b2 = taylm(a2, 6); % Taylor model with maximum order of 6 and name a2
b3 = taylm(a3, 6); % Taylor model with maximum order of 6 and name a3
b4 = taylm(a4, 6); % Taylor model with maximum order of 6 and name a4

B1 = [b1; b2] % generate a row of Taylor models
B2 = [b3; b4] % generate a row of Taylor models

B1 + B2 % addition
B1' * B2 % matrix multiplication
B1 .* B2 % pointwise multiplication
B1 / 2 % division by scalar
B1 ./ B2 % pointwise division
B1.^3 % power function
sin(B1) % sine function
sin(B1(1,1)) + B1(2,1).^2 - B1' * B2  % combination of functions
```

The resulting workspace output is:

```
B1 =
         0.5 + 1.5*a1 + [0.00000,0.00000]
         2.5 + 0.5*a2 + [0.00000,0.00000]

B2 =
         -5.0 + a3 + [0.00000,0.00000]
         5.0 + a4 + [0.00000,0.00000]

B1 + B2 =
         -4.5 + 1.5*a1 + a3 + [0.00000,0.00000]
         7.5 + 0.5*a2 + a4 + [0.00000,0.00000]

B1' * B2 =
         10.0 - 7.5*a1 + 2.5*a2 + 0.5*a3 + 2.5*a4 + 1.5*a1*a3 + 0.5*a2*a4 + [0.00000,0.00000]

B1 .* B2 =
         -2.5 - 7.5*a1 + 0.5*a3 + 1.5*a1*a3 + [0.00000,0.00000]
         12.5 + 2.5*a2 + 2.5*a4 + 0.5*a2*a4 + [0.00000,0.00000]

B1 / 2 =
         0.25 + 0.75*a1 + [0.00000,0.00000]
         1.25 + 0.25*a2 + [0.00000,0.00000]

B1 ./ B2 =
         -0.1 - 0.3*a1 - 0.02*a3 - 0.06*a1*a3 - 0.004*a3^2 - 0.012*a1*a3^2
         - 0.0008*a3^3 - 0.0024*a1*a3^3 - 0.00016*a3^4 - 0.00048*a1*a3^4
         - 0.000032*a3^5 - 0.000096*a1*a3^5 - 6.4e-6*a3^6 + [-0.00005,0.00005]

         0.5 + 0.1*a2 - 0.1*a4 - 0.02*a2*a4 + 0.02*a4^2 + 0.004*a2*a4^2
         - 0.004*a4^3 - 0.0008*a2*a4^3 + 0.0008*a4^4 + 0.00016*a2*a4^4
         - 0.00016*a4^5 - 0.000032*a2*a4^5 + 0.000032*a4^6 + [-0.00005,0.00005]

B1.^3 =
```

```
          0.125 + 1.125*a1 + 3.375*a1^2 + 3.375*a1^3 + [0.00000,0.00000]
          15.625 + 9.375*a2 + 1.875*a2^2 + 0.125*a2^3 + [0.00000,0.00000]

sin(B1) =
          0.47943 + 1.3164*a1 - 0.53935*a1^2 - 0.49364*a1^3 + 0.10113*a1^4
          + 0.055535*a1^5 - 0.0075847*a1^6 + [-0.00339,0.00339]

          0.59847 - 0.40057*a2 - 0.074809*a2^2 + 0.01669*a2^3 + 0.0015585*a2^4
          - 0.00020863*a2^5 - 0.000012988*a2^6 + [-0.00000,0.00000]

sin(B1(1,1)) + B1(2,1).^2 - B1' * B2 =
          -3.2706 + 8.8164*a1 - 0.5*a3 - 2.5*a4 - 0.53935*a1^2 + 0.25*a2^2
          - 1.5*a1*a3 - 0.5*a2*a4 - 0.49364*a1^3 + 0.10113*a1^4
          + 0.055535*a1^5 - 0.0075847*a1^6 + [-0.00339,0.00339]
```

### 10.1.14 Affine

The following MATLAB code demonstrates how to use affine arithmetics in CORA (see Sec. 2.2.3.2):

```
% create affine object
I = interval(-1,1);
aff = affine(I);

% create taylor model object (for comparison)
maxOrder = 1;
tay = taylm(int,maxOrder,'x');

% define function
f = @(x) sin(x) * (x+1);

% evaluate the function with affine arithmetic and taylor model
intAff = interval(f(aff))
intTay = interval(f(tay))
```

The resulting workspace output is:

```
intAff =
 [-1.84147,2.84147]
intTay =
 [-1.84147,2.84147]
```

### 10.1.15 Zoo

The following MATLAB code demonstrates how to use the class `zoo` in CORA (see Sec. 2.2.3.3):

```
% create zoo object
I = interval(-1,1);
methods = {'interval','taylm(int)'};
maxOrder = 3;
z = zoo(I,methods,maxOrder);

% create taylor model object (for comparison)
maxOrder = 10;
tay = taylm(I,maxOrder,'x');

% define function
f = @(x) sin(x) * (x+1);

% evaluate the function with zoo-object and taylor model
```

```
intZoo = interval(f(z))
intTay = interval(f(tay))
```

The resulting workspace output is:

```
intZoo =
 [-1.34206,1.68294]
intTay =
 [-1.34207,2.18354]
```

## 10.2 Matrix Set Representations

In this section we present examples for set-based computation using the different matrix set representations in Sec. 3.

### 10.2.1 Matrix Polytopes

The following MATLAB code demonstrates some of the introduced methods:

```
P1(:,:,1) = [1 2; 3 4]; % 1st vertex of matrix polytope P1
P1(:,:,2) = [2 2; 3 3]; % 2nd vertex of matrix polytope P1
matP1 = matPolytope(P1); % instantiate matrix polytope P1

P2(:,:,1) = [-1 2; 2 -1]; % 1st vertex of matrix polytope P2
P2(:,:,2) = [-1 1; 1 -1]; % 2nd vertex of matrix polytope P2
matP2 = matPolytope(P2); % instantiate matrix polytope P2

matP3 = matP1 + matP2 % perform Minkowski addition and display result
matP4 = matP1 * matP2 % compute multiplication of and display result

intP = intervalMatrix(matP1) % compute interval matrix and display result
```

This produces the workspace output

```
matP3 =

dimension:
     2     2

nr of vertices:
     4

vertices:

(:,:,1) =

     0     4
     5     3


(:,:,2) =

     1     4
     5     2
```

```
(:,:,3) =

     0     3
     4     3


(:,:,4) =

     1     3
     4     2


matP4 =

dimension:
     2     2

nr of vertices:
     4

vertices:

(:,:,1) =

     3     0
     5     2


(:,:,2) =

     2     2
     3     3


(:,:,3) =

     1    -1
     1    -1


(:,:,4) =

     0     0
     0     0


intP =

intervalMatrix:
```

```
- dimension: 2  2

  [1, 2] [2, 2]
  [3, 3] [3, 4]
```

### 10.2.2 Matrix Zonotopes

The following MATLAB code demonstrates some of the introduced methods:

```
Zcenter = [1 2; 3 4]; % center of matrix zonotope Z1
Zdelta(:,:,1) = [1 0; 1 1]; % generators of matrix zonotope Z1
matZ1 = matZonotope(Zcenter, Zdelta); % instantiate matrix zonotope Z1

Zcenter = [-1 2; 2 -1]; % center of matrix zonotope Z2
Zdelta(:,:,1) = [0 0.5; 0.5 0]; % generators of matrix zonotope Z2
matZ2 = matZonotope(Zcenter, Zdelta); % instantiate matrix zonotope Z2

matZ3 = matZ1 + matZ2 % perform Minkowski addition and display result
matZ4 = matZ1 * matZ2 % compute multiplication of and display result

intZ = intervalMatrix(matZ1) % compute interval matrix and display result
```

This produces the workspace output

```
matZ3 =

matZonotope:
dimension:
     2     2

center:
     0     4
     5     3

generators: (2 generators)

(:,:,1) =

     1     0
     1     1


(:,:,2) =

          0    0.5000
     0.5000         0


matZ4 =

matZonotope:
dimension:
     2     2
```

```
center:
     3     0
     5     2

generators: (3 generators)

(:,:,1) =

    -1     2
     1     1


(:,:,2) =

    1.0000    0.5000
    2.0000    1.5000


(:,:,3) =

         0    0.5000
    0.5000    0.5000


intZ =

intervalMatrix:
- dimension: 2  2

  [0, 2] [2, 2]
  [2, 4] [3, 5]
```

### 10.2.3 Interval Matrices

The following MATLAB code demonstrates some of the introduced methods:

```
1  Mcenter = [1 2; 3 4]; % center of interval matrix M1
2  Mdelta = [1 0; 1 1]; % delta of interval matrix M1
3  intM1 = intervalMatrix(Mcenter, Mdelta); % instantiate interval matrix M1
4
5  Mcenter = [-1 2; 2 -1]; % center of interval matrix M2
6  Mdelta = [0 0.5; 0.5 0]; % delta of interval matrix M2
7  intM2 = intervalMatrix(Mcenter, Mdelta); % instantiate interval matrix M2
8
9  intM3 = intM1 + intM2 % perform Minkowski addition and display result
10 intM4 = intM1 * intM2 % compute multiplication of and display result
11
12 matZ = matZonotope(intM1) % compute matrix zonotope and display result
```

This produces the workspace output

```
intM3 =
```

```
intervalMatrix:
- dimension: 2  2

  [-1, 1] [3.5000, 4.5000]
  [3.5000, 6.5000] [2, 4]


intM4 =

intervalMatrix:
- dimension: 2  2

  [1, 5] [-2, 3]
  [0.5000, 10.5000] [-2, 7]


matZ =

matZonotope:
dimension:
     2     2

center:
     1     2
     3     4

generators: (4 generators)

(:,:,1) =

     1     0
     0     0


(:,:,2) =

     0     0
     1     0


(:,:,3) =

     0     0
     0     0


(:,:,4) =

     0     0
     0     1
```

## 10.3 Continuous Dynamics

This section presents a variety of examples for continuous dynamics categorized along the different classes for dynamic systems realized in CORA. All subsequent examples can handle uncertain inputs. Uncertain parameters can be realized using different techniques:

1. Introduce constant parameters as additional states and assign the dynamics $\dot{x}_i = 0$ to them. The disadvantage is that the dimension of the system is growing.
2. Introduce time-varying parameters as additional uncertain inputs.
3. Use specialized functions in CORA that can handle uncertain parameters.

It is generally advised to use the last technique, but there is no proof that this technique always provides better results compared to the other techniques.

### 10.3.1 Linear Dynamics

For linear dynamics we demonstrate the usage of two different reachability algorithms.

#### Standard Algorithm

First, we consider the standard algorithm from [32]. We use a simple academic example from [33, Sec. 3.2.3] with not much focus on a connection to a real system. However, since linear systems are solely determined by their state and input matrix, adjusting this example to any other linear system is straightforward. Here, the system dynamics is

$$\dot{x} = \begin{bmatrix} -1 & -4 & 0 & 0 & 0 \\ 4 & -1 & 0 & 0 & 0 \\ 0 & 0 & -3 & 1 & 0 \\ 0 & 0 & -1 & -3 & 0 \\ 0 & 0 & 0 & 0 & -2 \end{bmatrix} x + u(t), \quad x(0) \in \begin{bmatrix} [0.9, 1.1] \\ [0.9, 1.1] \\ [0.9, 1.1] \\ [0.9, 1.1] \\ [0.9, 1.1] \end{bmatrix}, u(t) \in \begin{bmatrix} [0.9, 1.1] \\ [-0.25, 0.25] \\ [-0.1, 0.1] \\ [0.25, 0.75] \\ [-0.75, -0.25] \end{bmatrix}.$$

The MATLAB code that implements the simulation and reachability analysis of the linear example is (see file *examples/contDynamics/linearSys/example_linear_reach_01_5dim.m* in the CORA toolbox):

```
% Parameter ---------------------------------------------------------------

params.tFinal = 5;
params.R0 = zonotope([ones(5,1),0.1*diag(ones(5,1))]);
params.U = zonotope(interval([0.9; -0.25; -0.1; 0.25; -0.75], ...
                             [1.1; 0.25; 0.1; 0.75; -0.25]));


% Reachability Settings ---------------------------------------------------

options.timeStep = 0.02;
options.taylorTerms = 4;
options.zonotopeOrder = 20;


% System Dynamics ---------------------------------------------------------

A = [-1 -4 0 0 0; 4 -1 0 0 0; 0 0 -3 1 0; 0 0 -1 -3 0; 0 0 0 0 -2];
B = 1;

fiveDimSys = linearSys('fiveDimSys',A,B);
```

```
% Reachability Analysis -------------------------------------------------

tic
R = reach(fiveDimSys, params, options);
tComp = toc;
disp(['computation time of reachable set: ',num2str(tComp)]);


% Simulation -------------------------------------------------------------

simOpt.points = 25;
simOpt.fracVert = 0.5;
simOpt.fracInpVert = 0.5;
simOpt.nrConstInp = 10;

simRes = simulateRandom(fiveDimSys, params, simOpt);


% Visualization ----------------------------------------------------------

% plot different projections
dims = {[1 2],[3 4]};

for k = 1:length(dims)

    figure; hold on
    projDims = dims{k};

    % plot reachable sets
    plot(R,projDims,'FaceColor',[.8 .8 .8],'EdgeColor','b');

    % plot initial set
    plot(params.R0,projDims,'w-','lineWidth',2);

    % plot simulation results
    plot(simRes,projDims,'y');

    % label plot
    xlabel(['x_{',num2str(projDims(1)),'}']);
    ylabel(['x_{',num2str(projDims(2)),'}']);
end
```

The reachable set and the simulation are plotted in Fig. 41 for a time horizon of $t_f = 5$.

### Adaptive Algorithm

One major disadvantage of the standard algorithm used in the example above is that the user is required to manually tune the time step size, the number of Taylor terms, and the zonotope order to obtain a tight enclosure of the reachable set. The novel adaptive algorithm from [69] tunes these parameters automatically in such a way that a certain precision is achieved. We consider the two-dimensional system

$$\dot{x} = \begin{bmatrix} -0.7 & -2 \\ 2 & -0.7 \end{bmatrix} x + u(t), \quad x(0) \in \begin{bmatrix} [9.5, 10.5] \\ [4.5, 5.5] \end{bmatrix}, u(t) \in \begin{bmatrix} [0.75, 1.25] \\ [0.75, 1.25] \end{bmatrix}.$$

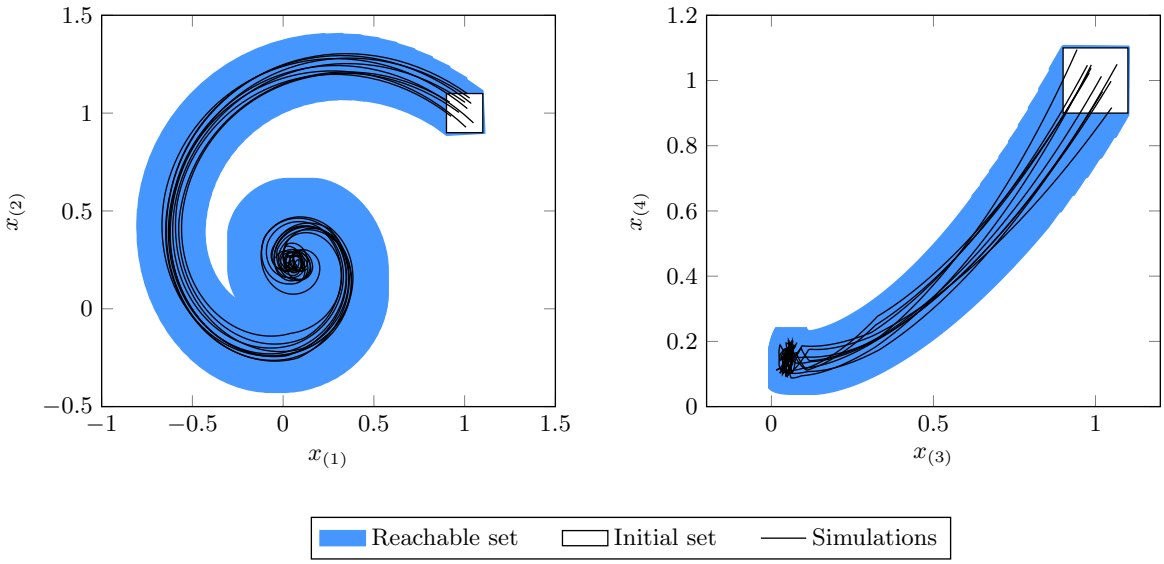



Figure 41: Illustration of the reachable set of the linear example. The white box shows the initial set and the black lines show simulated trajectories.

The MATLAB code that implements reachability analysis with the adaptive algorithm (see file *examples/contDynamics/linearSys/example_linear_reach_04_adaptive.m* in the CORA toolbox):

```
% System Dynamics ------------------------------------------------------------

A = [-0.7 -2; 2 -0.7];
B = 1;

sys = linearSys('sys',A,B);


% Parameter ------------------------------------------------------------------

dim = length(A);

params.tFinal = 5;
params.R0 = zonotope([[10; 5],0.5*eye(dim)]);       % initial set
params.U = zonotope([ones(dim,1),0.25*eye(dim)]);   % uncertain inputs


% Reachability Settings ------------------------------------------------------

options.linAlg = 'adaptive'; % adaptive parameter tuning


% Simulation -----------------------------------------------------------------

simOpt.points = 10;
simOpt.fracVert = 0.5;
simOpt.fracInpVert = 0.5;
simOpt.nrConstInp = 10;

simRes = simulateRandom(sys, params, simOpt);


% Reachability Analysis ------------------------------------------------------
```

```
errs = [1;0.05];
stepssS = zeros(length(errs),1);
timesS = zeros(length(errs),1);
R = cell(length(errs),1);

% compute reachable sets for different max. allowed errors
for i=1:length(errs)
    options.error = errs(i);
    tic
    R{i} = reach(sys,params,options);
    timesS(i) = toc;
    stepssS(i) = length(R{i}.timeInterval.set);
end


% Visualization ----------------------------------------------------------

figure; hold on; box on;
projDims = [1,2];

% plot reachable set
plot(R{1},projDims,'k','EdgeColor','k');
plot(R{2},projDims,'FaceColor',[0.7,0.7,0.7],'EdgeColor',[0.7,0.7,0.7]);

% plot initial set
plot(params.R0,projDims,'w','LineWidth',1.5);

% plot simulation
plot(simRes,projDims,'b','LineWidth',0.5);

% plot unsafe set
unsafeSet = interval([2;-2],[4;2]);
plot(unsafeSet,projDims,'FaceColor',[227,114,34]/255,...
    'EdgeColor','r','LineWidth',2);

% formatting
xlabel('x_1'); ylabel('x_2');
title('2D system');
```

The reachable sets computed with the adaptive algorithm for two different precision values are plotted in Fig. 42 for a time horizon of $t_f = 5$.

### 10.3.2 Linear Dynamics with Uncertain Parameters

For linear dynamics with uncertain parameters, we use the transmission line example from [118, Sec. 4.5.2], which can be modeled as an electric circuit with resistors, inductors, and capacitors. The parameters of each component have uncertain values as described in [118, Sec. 4.5.2]. This example shows how one can better take care of dependencies of parameters by using matrix zonotopes instead of interval matrices.

The MATLAB code that implements the simulation and reachability analysis of the linear example with uncertain parameters is (see file *examples/contDynamics/linParamSys/ example_linearParam_reach_01_rlc_const.m* in the CORA toolbox):

```
% System Dynamics -------------------------------------------------------
```

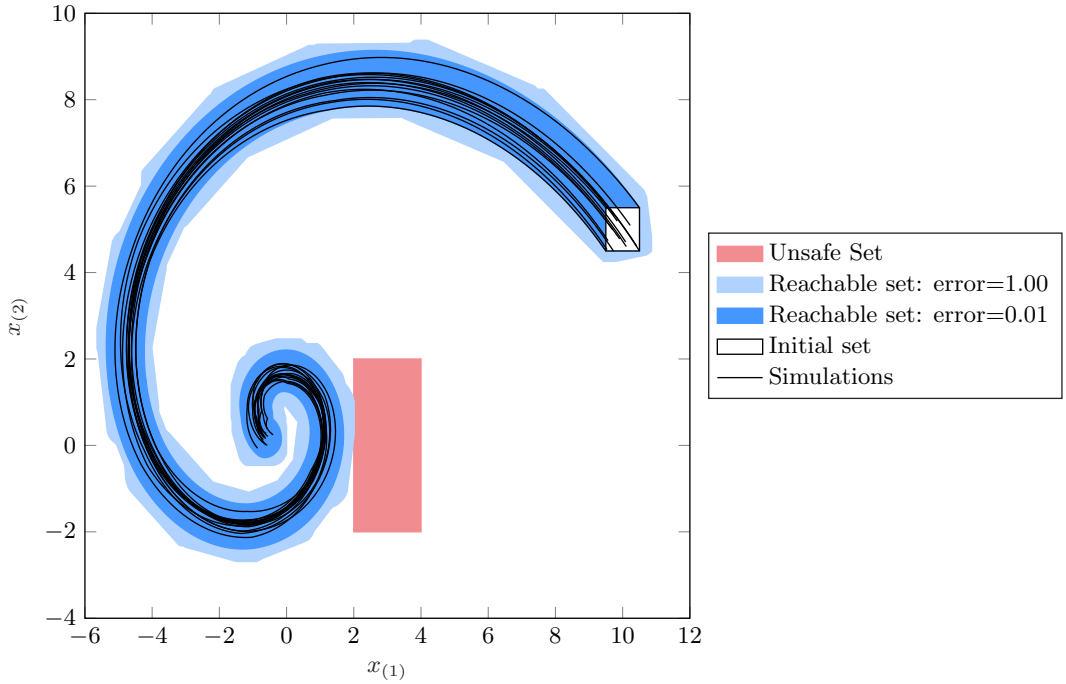



Figure 42: Illustration of the reachable set computed with the adaptive algorithm with a requested precision of `options.error = 1` (black) and `options.error = 0.05` (gray).

```
% get matrix zonotopes of the model
[matZ_A,matZ_B] = RLCcircuit();
matI_A = intervalMatrix(matZ_A);
dim = matZ_A.dim;

% create linear parametric systems with constant parameters
sysMatZono  = linParamSys(matZ_A, eye(dim));
sysIntMat = linParamSys(matI_A, eye(dim));


% Parameter ---------------------------------------------------------------

% compute initial set
u0 = intervalMatrix(0,0.2);             % range of voltages

intA = intervalMatrix(matZ_A);
invAmid = inv(center(intA.int));        % inverse of A

intB = intervalMatrix(matZ_B);
R0 = invAmid*intB*u0 + intervalMatrix(0,1e-3*ones(dim,1));

params.R0 = zonotope(interval(R0));     % convert initial set to zonotope

% uncertain inputs
u = intervalMatrix(1,0.01);
params.U = zonotope(interval(intB*u));

% final time
params.tFinal = 0.3;


% Reachability Settings ---------------------------------------------------
```

```
options.intermediateOrder = 2;
options.timeStep = 0.001;
options.zonotopeOrder = 400;
options.taylorTerms = 8;
options.compTimePoint = false;


% Reachability Analysis -------------------------------------------------

% compute reachable set using matrix zonotopes
tic
RmatZono = reach(sysMatZono, params, options);
tComp = toc;
disp(['computation time (matrix zonotopes): ',num2str(tComp)]);

% compute reachable set using interval matrices
tic
RintMat = reach(sysIntMat, params, options);
tComp = toc;
disp(['computation time (interval matrices): ',num2str(tComp)]);


% Simulation ------------------------------------------------------------

simOpt.points = 60;
simOpt.fracVert = 0.5;
simOpt.fracInpVert = 0.5;
simOpt.nrConstInp = 6;

simRes = simulateRandom(sysIntMat, params, simOpt);


% Visualization ---------------------------------------------------------

% PLOT 1: state space

figure;
hold on
projDim = [20,40];

% plot reachable sets
hanIntMat = plot(RintMat,projDim,'FaceColor',[.6 .6 .6],'Order',10);
hanMatZono = plot(RmatZono,projDim,'FaceColor',[.8 .8 .8],'Order',10);

% plot initial set
plot(params.R0,projDim,'k','FaceColor','k');

% plot simulation results
plot(simRes,projDim);

% label plot
xlabel(['x_{',num2str(projDim(1)),'}']);
ylabel(['x_{',num2str(projDim(2)),'}']);
legend([hanIntMat,hanMatZono],'Interval matrix','Matrix zonotope');


% PLOT 2: reachable set over time
```

```
figure;
hold on

% plot time elapse
hanIntMat = plotOverTime(RintMat,0.5*dim,'FaceColor',[.6 .6 .6]);
hanMatZono = plotOverTime(RmatZono,0.5*dim,'FaceColor',[.8 .8 .8]);

% plot simulation results
plotOverTime(simRes,0.5*dim);

% label plot
xlabel('t');
ylabel(['x_{',num2str(0.5*dim),'}']);
legend([hanIntMat,hanMatZono],'Interval matrix','Matrix zonotope');
```

The reachable set and the simulation are plotted in Fig. 43 for a time horizon of $t_f = 0.3$.

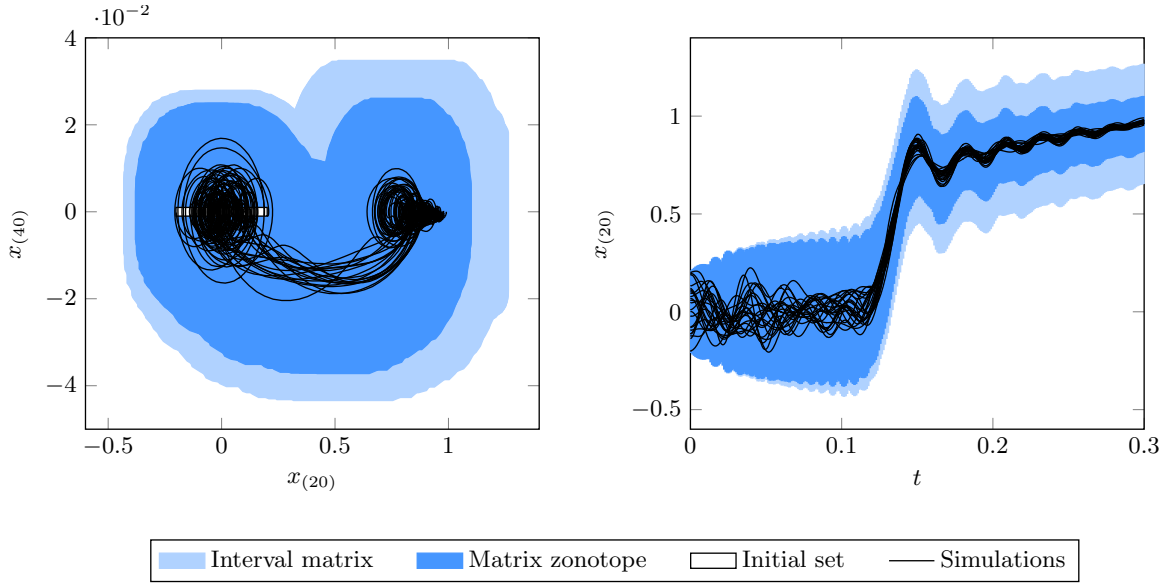



Figure 43: Illustration of the reachable set of the transmission example. A white box shows the initial set and the black lines are simulated trajectories.

### 10.3.3 Nonlinear Dynamics

For nonlinear dynamics, several examples are presented.

#### Tank System

The first example is the tank system from [15] where water flows from one tank into another one. This example can be used to study the effect of water power plants on the water level of rivers. This example can be easy extended by several tanks and thus is a nice benchmark example to study the scalability of algorithms for reachability analysis. CORA can compute the reachable set with at least 100 tanks.

The MATLAB code that implements the simulation and reachability analysis of the tank example is (see file *examples/contDynamics/nonlinearSys/ example_nonlinear_reach_01_tank.m* in the CORA toolbox):

```
% Parameter ---------------------------------------------------------------

params.tFinal = 400;
params.R0 = zonotope([[2; 4; 4; 2; 10; 4],0.2*eye(6)]);
params.U = zonotope([0,0.005]);


% Reachability Settings ----------------------------------------------------

options.timeStep = 1;
options.taylorTerms = 4;
options.zonotopeOrder = 50;

options.intermediateOrder = 5;
options.errorOrder = 1;
options.alg = 'lin';
options.tensorOrder = 2;


% System Dynamics ----------------------------------------------------------

tank = nonlinearSys(@tank6Eq);


% Reachability Analysis ----------------------------------------------------

tic
R = reach(tank, params, options);
tComp = toc;
disp(['computation time of reachable set: ',num2str(tComp)]);


% Simulation ---------------------------------------------------------------

simOpt.points = 60;
simOpt.fracVert = 0.5;
simOpt.fracInpVert = 0.5;
simOpt.nrConstInp = 6;

simRes = simulateRandom(tank, params, simOpt);


% Visualization ------------------------------------------------------------

dims = {[1 2],[3 4],[5 6]};

for k = 1:length(dims)

    figure; hold on
    projDim = dims{k};

    % plot reachable sets
    plot(R,projDim,'FaceColor',[.8 .8 .8]);

    % plot initial set
    plot(params.R0,projDim,'k','FaceColor','w');

    % plot simulation results
```

```
    plot(simRes,projDim,'k');

    % label plot
    xlabel(['x_{',num2str(projDim(1)),'}']);
    ylabel(['x_{',num2str(projDim(2)),'}']);
end
```

The difference to specifying a linear system is that a link to a nonlinear differential equation has to be provided, rather than the system matrix $A$ and the input matrix $B$. The nonlinear system model $\dot{x} = f(x, u)$, where $x$ is the state and $u$ is the input, is shown below:

```
function dx = tank6Eq(x,u)

    % parameter
    k = 0.015;
    k2 = 0.01;
    g = 9.81;

    % differential equations
    dx(1,1) = u(1)+0.1+k2*(4-x(6))-k*sqrt(2*g)*sqrt(x(1));   % tank 1
    dx(2,1) = k*sqrt(2*g)*(sqrt(x(1))-sqrt(x(2)));           % tank 2
    dx(3,1) = k*sqrt(2*g)*(sqrt(x(2))-sqrt(x(3)));           % tank 3
    dx(4,1) = k*sqrt(2*g)*(sqrt(x(3))-sqrt(x(4)));           % tank 4
    dx(5,1) = k*sqrt(2*g)*(sqrt(x(4))-sqrt(x(5)));           % tank 5
    dx(6,1) = k*sqrt(2*g)*(sqrt(x(5))-sqrt(x(6)));           % tank 6
```

The output of this function is $\dot{x}$ for a given time $t$, state $x$, and input $u$.

Fig. 44 shows the reachable set and the simulation for a time horizon of $t_f = 400$.

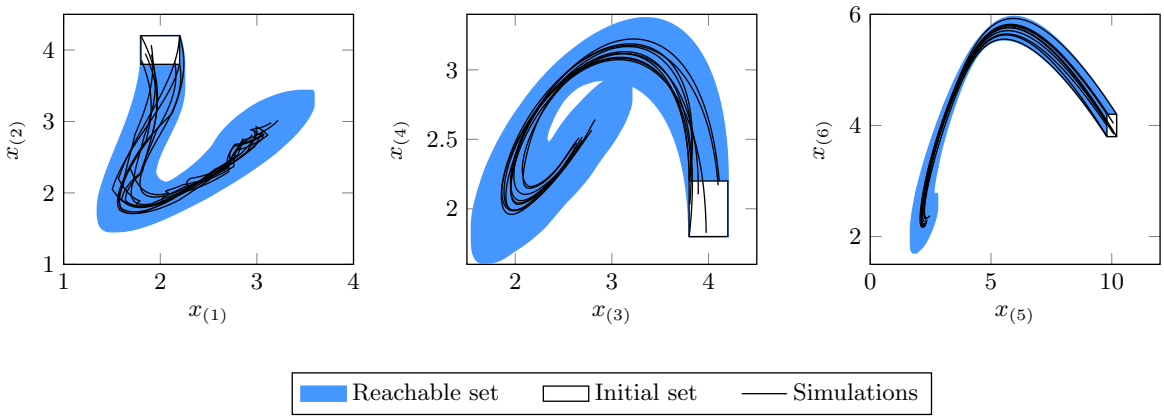



Figure 44: Illustration of the reachable set of the linear example. The white box shows the initial set and the black lines show simulated trajectories.

### Van der Pol Oscillator

The Van der Pol oscillator is a standard example for limit cycles. By using reachability analysis one can show that one always returns to the initial set so that the obtained set is an invariant set. This example is used in [15] to demonstrate that one can obtain a solution even if the linearization error becomes too large by splitting the reachable set. Later, in [38] an improved method is presented that requires less splitting. This example demonstrates the capabilities of the simpler approach presented in [15]. Due to the similarity of the

MATLAB code compared to the previous tank example, we only present the reachable set in Fig. 45. The corresponding code can be found in the file *examples/contDynamics/nonlinearSys/ example_nonlinear_reach_03_vanDerPol_splitting.m* in the CORA toolbox.

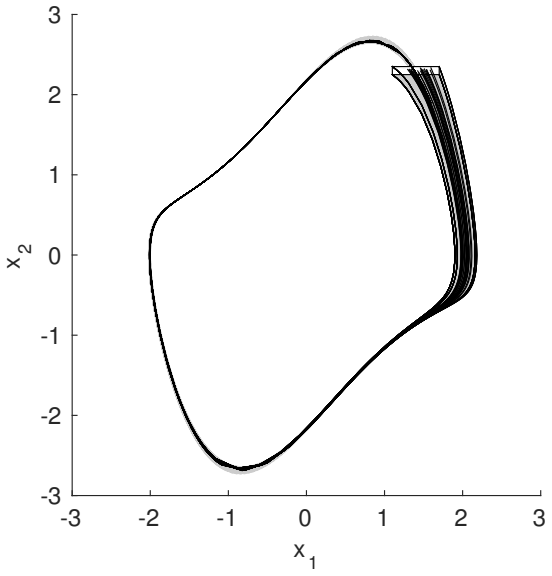



Figure 45: Illustration of the reachable set of the Van der Pol oscillator. The white box shows the initial set and the black lines show simulated trajectories.

**Seven-Dimensional Example for Non-Convex Set Representation**

This academic example is used to demonstrate the benefits of using higher-order abstractions of nonlinear systems compared to linear abstractions. However, since higher order abstractions do not preserve convexity when propagating reachable sets, the non-convex set representation *polynomial zonotope* is used as presented in [38]. Please note that the entire reachable set for the complete time horizon is typically non-convex, even when the propagation from one point in time to another point in time is convex. Due to the similarity of the MATLAB code compared to the previous tank example, we only present the reachable set in Fig. 46. The corresponding code can be found in the file *examples/contDynamics/nonlinearSys/ example_nonlinear_reach_04_laubLoomis_polyZonotope.m* in the CORA toolbox.

**Autonomous Car Following a Reference Trajectory**

This example presents the reachable set of an automated vehicle developed at the German Aerospace Center. The difference of this example compared to the previous example is that a reference trajectory is followed. Similar models have been used in previous publications, see e.g., [60, 119, 120]. In CORA, this only requires changing the input in `options.uTrans` from a vector to a matrix, where each column vector is the reference value at the next sampled point in time. Due to the similarity of the MATLAB code compared to the previous tank example, we only present the reachable set in Fig. 47, where the reference trajectory is plotted in red. The corresponding code can be found in the file *examples/contDynamics/nonlinearSys/ example_nonlinear_reach_05_autonomousCar.m* in the CORA toolbox.

### 10.3.4 Nonlinear Dynamics with Uncertain Parameters

As for linear systems, specialized algorithms have been developed for considering uncertain parameters of nonlinear systems. To better compare the results, we again use the tank system

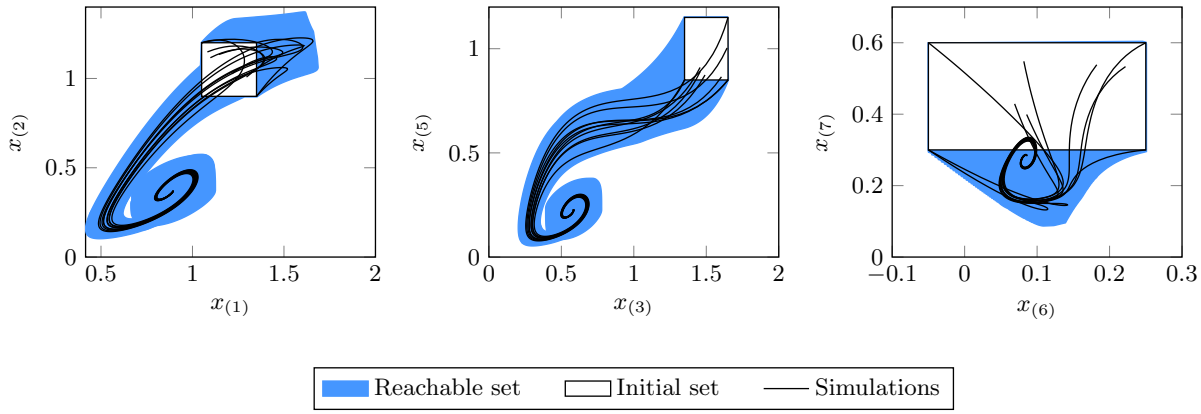



Figure 46: Illustration of the reachable set of the seven-dimensional example for non-convex set representation. The white box shows the initial set and the black lines show simulated trajectories.

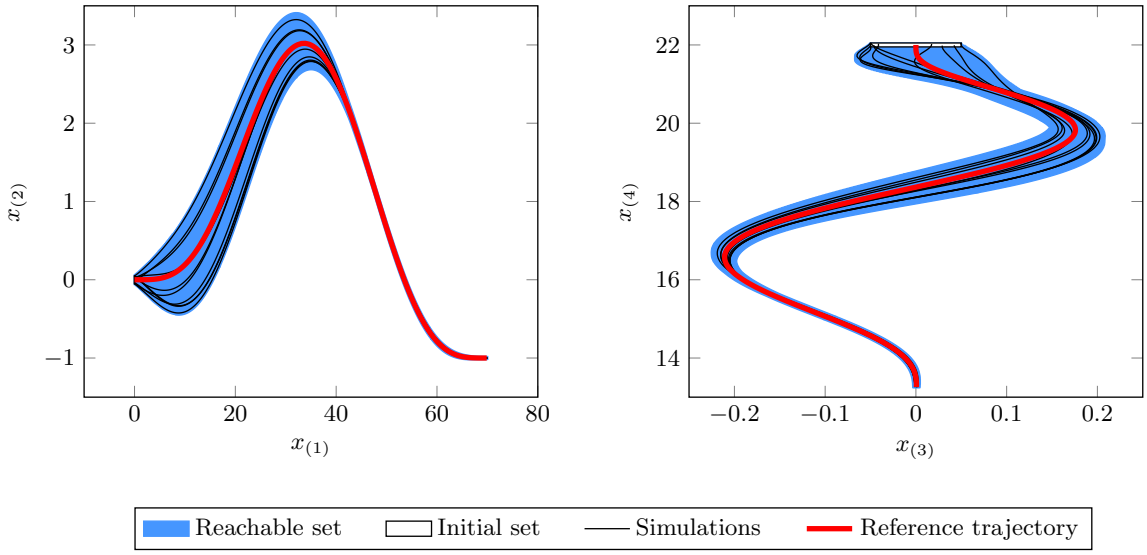



Figure 47: Illustration of the reachable set of the seven-dimensional example for non-convex set representation. The white box shows the initial set and the black lines show simulated trajectories.

whose reachable set we know from a previous example. The plots show not only the case with uncertain parameters, but also the one without uncertain parameters.

The MATLAB code that implements the simulation and reachability analysis of the nonlinear example with uncertain parameters is (see file *examples/contDynamics/nonlinParamSys/example_nonlinearParam_reach_01_tank.m* in the CORA toolbox):

```
% Parameter ---------------------------------------------------------------

params.tFinal = 400;                                        % final time
params.R0 = zonotope([[2; 4; 4; 2; 10; 4],0.2*eye(6)]);     % initial set
params.U = zonotope([0,0.005]);                             % uncertain input


% Reachability Settings ---------------------------------------------------

options.timeStep=0.5;
```

```
options.taylorTerms=4;
options.intermediateOrder = 4;
options.zonotopeOrder=10;
options.tensorOrder = 2;
options.alg = 'lin';


% System Dynamics ------------------------------------------------------

% tank system with certain pararmters
tank = nonlinearSys(@tank6Eq);

% tank system with uncertain parameters
optionsParam = options;
optionsParam.paramInt = interval(0.0148,0.015);

tankParam = nonlinParamSys(@tank6paramEq);


% Reachability Analysis ------------------------------------------------

% compute reachable set of tank system without uncertain parameters
tic
RcontNoParam = reach(tank, params, options);
tComp = toc;
disp(['computation time (without uncertain parameters): ',num2str(tComp)]);

% compute reachable set of tank system with uncertain parameters
tic
RcontParam = reach(tankParam, params, optionsParam);
tComp = toc;
disp(['computation time (with uncertain parameters): ',num2str(tComp)]);


% Simulation -----------------------------------------------------------

% settings for random simulation
simOpt.points = 60;         % number of initial points
simOpt.fracVert = 0.5;      % fraction of vertices initial set
simOpt.fracInpVert = 0.5;   % fraction of vertices input set
simOpt.nrConstInp = 6;      % changes of input over time horizon

% random simulation
simRes = simulateRandom(tank,params,simOpt);


% Visualization --------------------------------------------------------

dims = {[1,2],[3,4],[5,6]};

% plot different projections
for i = 1:length(dims)

    figure; hold on
    projDims = dims{i};

    % plot reachable sets
    hanParam = plot(RcontParam,projDims,'FaceColor',[.7 .7 .7])
```

```
    hanNoParam = plot(RcontNoParam,projDims,'w');

    % plot initial set
    plot(params.R0,projDims,'k','FaceColor','w');

    % plot simulation results
    plot(simRes,projDims);

    % label plot
    xlabel(['x_{',num2str(projDims(1)),'}']);
    ylabel(['x_{',num2str(projDims(2)),'}']);
end
```

The reachable set and the simulation are plotted in Fig. 48 for a time horizon of $t_f = 400$.

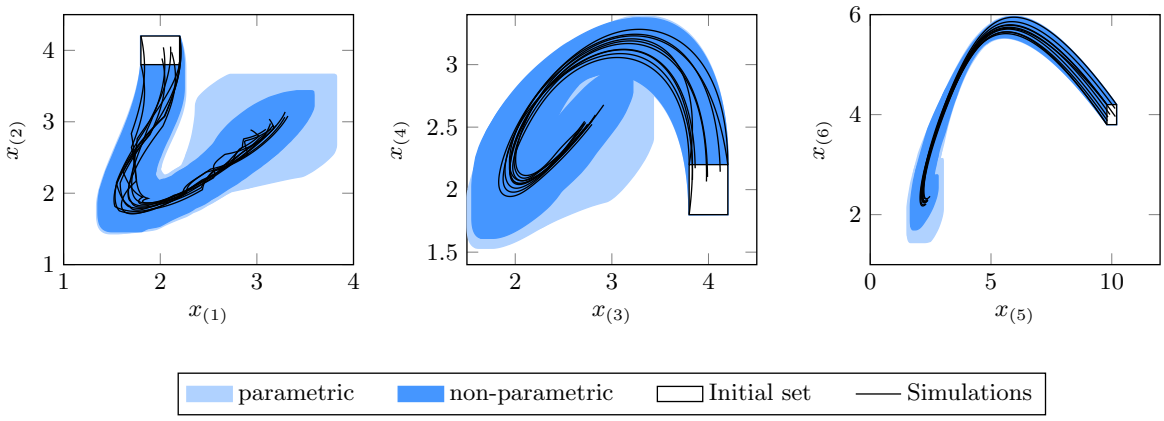



Figure 48: Illustration of the reachable set of the nonlinear parametric example. The light-blue region shows the reachable set with uncertain parameters, while the blue area shows the reachable set without uncertain parameters. The white box shows the initial set and the black lines show simulated trajectories.

### 10.3.5 Discrete-time Nonlinear Systems

We demonstrate the calculation of the reachable set for a time-discrete system with the example of a stirred tank reactor model. The original continuous time system model is given in [121]. Using the trapezoidal rule for time discretization, we obtained the following nonlinear discrete time system:

$$C_A(k+1) = \frac{1 - \frac{q\tau}{2V} - k_0 \cdot \tau \cdot \exp\left(-\frac{E}{R \cdot T(k)}\right) \cdot C_A(k) + \frac{q}{V} \cdot C_{Af} \cdot \tau}{1 + \frac{q\tau}{2V} + w_1(k) \cdot \tau}$$

$$T(k+1) = \frac{T(k) \cdot \left(1 - \frac{\tau}{2} - \frac{\tau \cdot UA}{2V \cdot \rho \cdot C_p}\right) + \tau \cdot \left(T_f \cdot \frac{q}{V} + \frac{UA \cdot u(C_A(k), T(k))}{V \cdot \rho \cdot C_p}\right)}{1 + \frac{\tau \cdot q}{2V} + \frac{\tau \cdot UA}{2V \cdot \rho \cdot C_p}} \tag{55}$$

$$- \frac{C_A(k) \cdot \frac{\Delta H \cdot k_0 \cdot \tau}{\rho \cdot C_p} \cdot \exp\left(-\frac{E}{R \cdot T(k)}\right)}{1 + \frac{\tau \cdot q}{2V} + \frac{\tau \cdot UA}{2V \cdot \rho \cdot C_p}} + \tau \cdot w_2(k) \quad ,$$

where $u(C_A(k), T(k)) = -3 \cdot C_A(k) - 6.9 \cdot T(k)$ is the linear control law, $w_1(k) \in [-0.1, 0.1]$ and $w_2(k) \in [-2, 2]$ are bounded disturbances, and $\tau$ is the time step size. The values for the

model parameters are given in [121]. The MATLAB code that implements the simulation and reachability analysis for the nonlinear discrete time model is shown below (see file *examples/contDynamics/nonlinearSysDT/example_nonlinearDT_reach_cstrDisc.m* in the CORA toolbox):

```
% Parameter ---------------------------------------------------------

params.tFinal = 0.15;
params.R0 = zonotope([[-0.15;-45],diag([0.005;3])]);
params.U = zonotope([zeros(2,1),diag([0.1;2])]);


% Reachability Settings  --------------------------------------------

options.zonotopeOrder = 100;
options.tensorOrder = 3;
options.errorOrder = 5;


% System Dynamics  --------------------------------------------------

% sampling time
dt = 0.015;

fun = @(x,u) cstrDiscr(x,u,dt);

sysDisc = nonlinearSysDT('stirredTankReactor',fun,0.015);


% Reachability Analysis ---------------------------------------------

tic
R = reach(sysDisc,params,options);
tComp = toc;
disp("Computation time: " + tComp);


% Simulation --------------------------------------------------------

simOpt.points = 100;
simOpt.fracVert = 0.5;
simOpt.fracInpVert = 0.5;
simOpt.nrConstInp = 3;

simRes = simulateRandom(sysDisc, params, simOpt);


% Visualization -----------------------------------------------------

figure; hold on; box on;

% plot initial set
plot(params.R0,[1,2],'FaceColor',[.8 .8 .8]);

% plot reachable set
plot(R,[1 2],'FaceColor',[.8 .8 .8]);

% plot simulation
plot(simRes,[1,2],'.k');
```

```
% formatting
xlabel('T-T_0');
ylabel('C-C_0');
```

The reachable set and the simulation are displayed in Fig. 49 for a time horizon of $t_f = 0.15$ min.

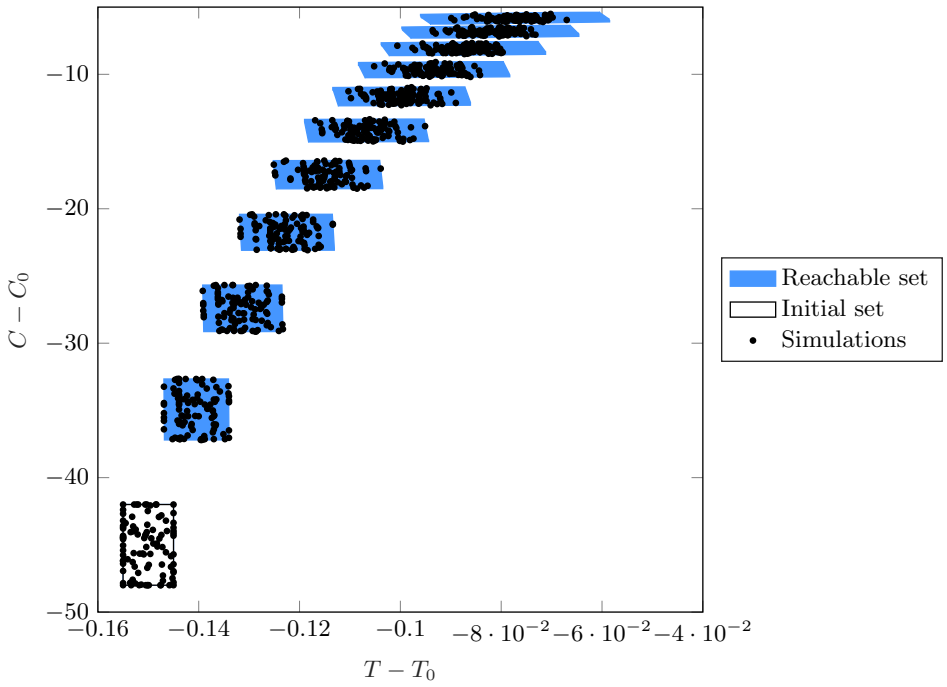



Figure 49: Illustration of the reachable set of the nonlinear discrete-time example. The black dots show the simulated points.

### 10.3.6 Nonlinear Differential-Algebraic Systems

CORA is also capable of computing reachable sets for semi-explicit, index-1 differential-algebraic equations. Although many index-1 differential-algebraic equations can be transformed into an ordinary differential equation, this is not always possible. For instance, power systems cannot be simplified due to Kirchhoff's law which constraints the currents of a node to sum up to zero. The capabilities of computing reachable sets are demonstrated for a small power system consisting of three buses. More complicated examples can be found in [96, 122, 123].

The MATLAB code that implements the simulation and reachability analysis of the nonlinear differential-algebraic example is (see file *examples/contDynamics/nonlinDASys/ example_nonlinearDA_reach_01_powerSystem_3bus.m* in the CORA toolbox):

```
% Parameter ------------------------------------------------------------

nrOfConstr = 6;
params.tFinal = 5;

x0 = [380; 0.7];
params.y0guess = [ones(0.5*nrOfConstr, 1); zeros(0.5*nrOfConstr, 1)];
params.R0 = zonotope([x0,diag([0.1, 0.01])]);

params.U = zonotope([[1; 0.4],diag([0, 0.04])]);
```

```
% Reachability Settings ------------------------------------------------

options.timeStep = 0.05;
options.taylorTerms = 6;
options.zonotopeOrder = 200;
options.errorOrder = 1.5;
options.tensorOrder = 2;

options.maxError = [0.5; 0];
options.maxError_x = options.maxError;
options.maxError_y = 0.005*[1; 1; 1; 1; 1; 1];


% System Dynamics -------------------------------------------------------

powerDyn = nonlinDASys(@bus3Dyn,@bus3Con);


% Reachability Analysis -------------------------------------------------

tic
R = reach(powerDyn, params, options);
tComp = toc;
disp(['computation time of reachable set: ',num2str(tComp)]);


% Simulation ------------------------------------------------------------

simOpt.points = 60;
simOpt.fracVert = 0.5;
simOpt.fracInpVert = 0.5;
simOpt.nrConstInp = 6;

simRes = simulateRandom(powerDyn, params, simOpt);


% Visualization ---------------------------------------------------------

dim = [1 2];

figure; hold on

% plot reachable sets
plot(R,dim,'FaceColor',[.7 .7 .7]);

% plot initial set
plot(params.R0,dim,'k','FaceColor','w');

% plot simulation results
plot(simRes,dim);

% label plot
xlabel(['x_{',num2str(dim(1)),'}']);
ylabel(['x_{',num2str(dim(2)),'}']);
```

The reachable set and the simulation are plotted in Fig. 50 for a time horizon of $t_f = 5$.

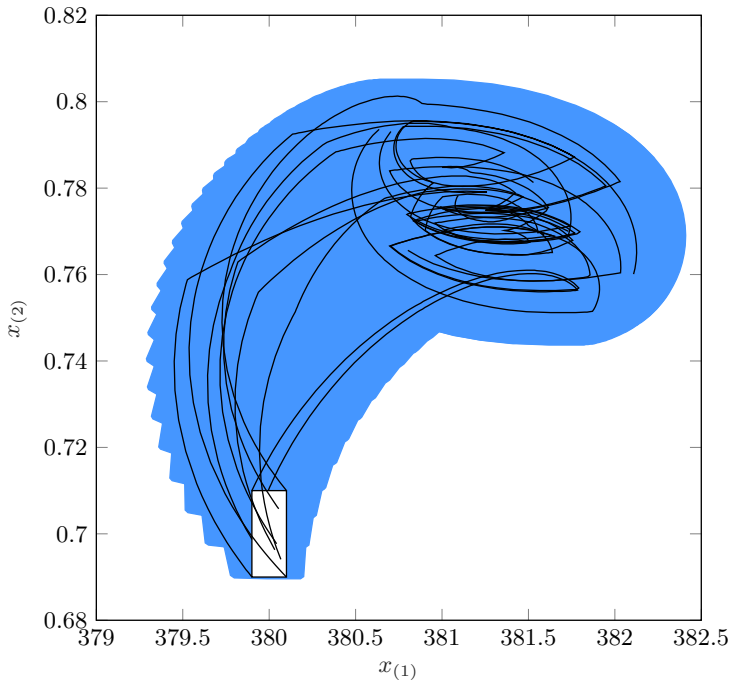


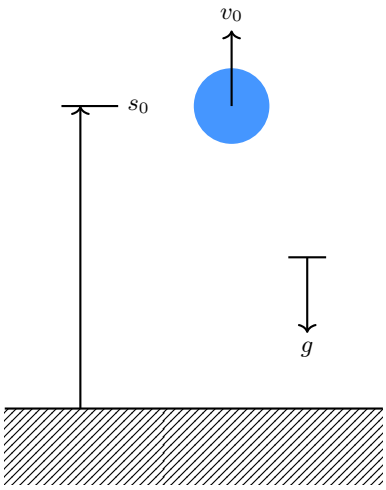

Figure 50: Illustration of the reachable set of nonlinear differential-algebraic example. The white box shows the initial set and the black lines show simulated trajectories.

## 10.4 Hybrid Dynamics

As already described in Sec. 4.3, CORA can compute reachable sets of mixed discrete/continuous or so-called hybrid systems. The difficulty in computing reachable sets of hybrid systems is the intersection of reachable sets with guard sets and the subsequent enclosure by the used set representation. As demonstrated in Sec. 4.3.1.1, CORA implements multiple different approaches for handling intersections with guard sets, some of which are demonstrated by the examples shown here.

### 10.4.1 Bouncing Ball Example

We demonstrate the syntax of CORA for the well-known bouncing ball example, see e.g., [124, Section 2.2.3]. Given is a ball in Fig. 51 with dynamics $\ddot{s} = -g$, where $s$ is the vertical position and $g$ is the gravity constant. After impact with the ground at $s = 0$, the velocity changes to $v' = -\alpha v$ ($v = \dot{s}$) with $\alpha \in [0, 1]$. The corresponding hybrid automaton can be formalized according to Sec. 4.3 as




$$
\begin{aligned}
HA &= (L_1) \\
L_1 &= (f_1(\cdot), \mathcal{S}_1, (T_1)) \\
f_1(x,u) &= \begin{bmatrix} x_2 \\ -g \end{bmatrix}, \quad g = 9.81 \\
\mathcal{S}_1 &= \left\{ [x_1 \ x_2]^T \in \mathbb{R}^2 \;\middle|\; x_2 \geq 0 \right\} \\
T_1 &= (\mathcal{G}_1, r_1(\cdot), 1) \\
\mathcal{G}_1 &= \left\{ [x_1 \ x_2]^T \in \mathbb{R}^2 \;\middle|\; x_1 = 0, \ x_2 \leq 0 \right\} \\
r(x) &= \begin{bmatrix} x_1 \\ -\alpha x_2 \end{bmatrix}, \quad \alpha = 0.75
\end{aligned}
$$

Figure 51: Example for a hybrid system: bouncing ball.

The MATLAB code that implements the simulation and reachability analysis of the bouncing ball example is (see file *examples/hybridDynamics/hybridAutomaton/ example_hybrid_reach_01_bouncingBall.m* in the CORA toolbox):

```
% Parameter ---------------------------------------------------------------

% problem description
params.R0 = zonotope([1;0],diag([0.05,0.05]));      % initial set
params.startLoc = 1;                                % initial location
params.tFinal = 1.7;                                % final time


% Reachability Options ----------------------------------------------------

% settings for continuous reachability
options.timeStep = 0.05;
options.taylorTerms = 10;
options.zonotopeOrder = 20;

% settings for hybrid systems
options.guardIntersect = 'polytope';
options.enclose = {'box'};


% Hybrid Automaton --------------------------------------------------------

% continuous dynamics
A = [0 1; 0 0];
B = [0; 0];
c = [0; -9.81];
linSys = linearSys('linearSys',A,B,c);

% system parameters
alpha = -0.75;                  % rebound factor

% invariant set
inv = polytope([-1,0],0);

% guard sets
guard = polytope([0 1],0,[1 0],0);
```

```
% reset function
reset = linearReset([1 0; 0 alpha],[0;0],[0;0]);

% transitions
trans(1) = transition(guard,reset,1);

% location object
loc(1) = location('loc1',inv,trans,linSys);

% hybrid automata
HA = hybridAutomaton(loc);


% Reachability Analysis -------------------------------------------------

tic;
R = reach(HA,params,options);
tComp = toc;

disp(['Computation time for reachable set: ',num2str(tComp),' s']);


% Simulation ------------------------------------------------------------

% settings for random simulation
simOpt.points = 10;         % number of initial points
simOpt.fracVert = 0.5;      % fraction of vertices initial set
simOpt.fracInpVert = 0.5;   % fraction of vertices input set
simOpt.inpChanges = 10;     % changes of input over time horizon

% random simulation
simRes = simulateRandom(HA,params,simOpt);



% Visualization ---------------------------------------------------------

figure; hold on

% plot reachable set
plotOverTime(R,1,'b');

% plot initial set
plotOverTime(params.R0,1,'k','FaceColor','w');

% plot simulated trajectories
plotOverTime(simRes,1);
```

The reachable set and the simulation are plotted in Fig. 52 for a time horizon of $t_f = 1.7$.

### 10.4.2 Powertrain Example

The powertrain example is taken out of [101, Sec. 6], which models the powertrain of a car with backlash. To investigate the scalability of the approach, one can add further rotating masses, similarly to adding further tanks for the tank example. Since the code of the powertrain example is rather lengthy, we are not presenting it in the manual; the interested reader can look it up in the example folder of the CORA code. The reachable set and the simulation are

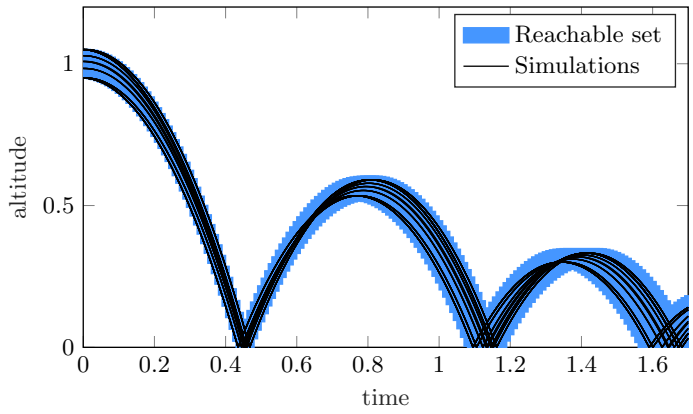



Figure 52: Illustration of the reachable set of the bouncing ball.

plotted in Fig. 53 for a time horizon of $t_f = 2$. The corresponding code can be found in the file *examples/hybridDynamics/hybridAutomaton/ example_hybrid_reach_02_powerTrain.m* in the CORA toolbox.

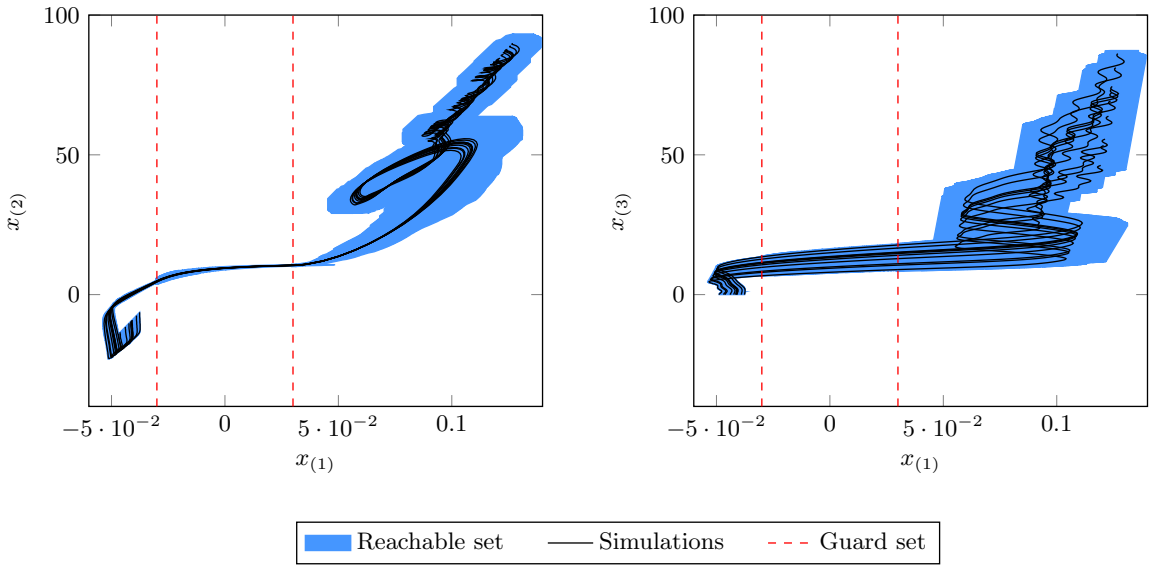



Figure 53: Illustration of the reachable set of the bouncing ball. The black box shows the initial set and the black line shows the simulated trajectory.

# 11 Conclusions

CORA is a toolbox for the implementation of prototype reachability analysis algorithms in MATLAB. The software is modular and is organized into four main categories: vector set representations, matrix set representations, continuous dynamics, and hybrid dynamics. CORA includes novel algorithms for reachability analysis of nonlinear systems and hybrid systems with a special focus on scalability; for instance, a power network with more than 50 continuous state variables has been verified in [123]. The efficiency of the algorithms used means it is even possible to verify problems online, i.e., while they are in operation [120].

One particularly useful feature of CORA is its adaptability: the algorithms can be tailored to the reachability analysis problem in question. Forthcoming integration into SpaceEx, which has a user interface and a model editor, should go some way towards making CORA more accessible to non-experts.

# Acknowledgments

The authors gratefully acknowledge financial support by the European Research Council (ERC) project justITSELF under grant agreement No 817629 as part of the EU Horizon 2020 program.

# A Additional Methods for Set Representations

In addition to the set operations described in Sec. 2.1, some set representations implement additional methods. This section documents most of the implemented methods and explains optional parameters for some methods.

## A.1 Zonotopes

In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5), the class `zonotope` supports the following methods:

- `abs` – returns a zonotope with absolute values of the center and the generators
- `box` – computes an enclosing axis-aligned box in generator representation.
- `constrSat` – checks if all values of a zonotope satisfy the constraint $Cx <= d$, $C \in \mathbb{R}^{m \times n}$, $d \in \mathbb{R}^m$.
- `deleteAligned` – combines aligned generators to a single generator. This reduces the order of a zonotope while not causing any over-approximation.
- `deleteZeros` – deletes generators whose entries are all zero.
- `dominantDirections` - computes the directions that span a parallelotope which tightly encloses a zonotope.
- `encloseMany` – function for the enclosure of multiple zonotopes with a zonotope.
- `enlarge` – enlarges the generators of a zonotope by a vector of factors for each dimension.
- `exactPlus` – compute the addition of two sets while preserving the dependencies between the two sets.
- `filterOut` – deletes parallelotopes that are covered by other parallelotopes from a list of parallelotopes
- `generatorLength` – returns the lengths of the generators.
- `generators` – returns the generators of a zonotope as a matrix whose column vectors are the generators.
- `intervalMultiplication` – multiplication of a zonotope with an interval (automatically called via mtimes)
- `intersectStrip` – encloses the intersection between a zonotope and a strip with a zonotope.
- `isInterval` – checks if a zonotope represents an interval.
- `minnorm` – returns the minimum zonotope norm.
- `minus` – approximates the Minkowski difference of two zonotopes or a zonotope and a vector.
- `norm` – computes the maximum norm value of all points in a zonotope. For more detail, see Appendix A.1.2.
- `orthVectors` – computes remaining orthogonal vectors when the zonotope is not full dimensional.
- `polygon` – converts a two-dimensional zonotope into a polygon and returns its vertices.

- `projectHighDim` – project a zonotope to a higher dimensional space.
- `quadMap_parallel` – parallel execution of quadMap-operation for zonotopes, see Sec. 2.1.1.5.
- `radius` – computes the radius of a hypersphere enclosing a zonotope.
- `rank` – computes the rank of the generator matrix.
- `reduceUnterApprox` – computes a zonotope with desired zonotope order which is a subset of the original zonotope.
- `rotate` – rotates a 2-dimensional zonotope by the specified angle.
- `sampleBox` – returns specified number of samples uniformly distributed in a full-dimensional parallelotope.
- `split` – splits a zonotope into two or more zonotopes that enclose the original zonotope. More details can be found in Appendix A.1.1.
- `splitFirstGen` – split a zonotope along the first generator.
- `tensorMultiplication` – computes $M_{ijk\ldots l}x_j x_k \cdots x_l | x \in \mathcal{Z}$ for a zonotope $\mathcal{Z}$ and a tensor $M$.
- `underapproximate` – returns the vertices of an under-approximation. The under-approximation is computed by finding the vertices that are extreme in the direction of a set of vectors, stored in the matrix S. If S is not specified, it is constructed by the vectors spanning an over-approximative parallelotope. (Warning: high computational complexity).
- `volumeRatio` – computes the approximate volume ratio of a zonotope and its over-approximating polytope
- `zonotopeNorm` – computes the norm of a point with respect to the zonotope-norm induced by the zonotope

### A.1.1 Method `split`

The ultimate goal is to compute the reachable set of a single point in time or time interval with a single set representation. However, reachability analysis often requires abstractions of the original dynamics, which might become inaccurate for large reachable sets. In that event it can be useful to split the reachable set and continue with two or more set representations for the same point in time or time interval. Zonotopes are not closed under intersection, and thus not under splits. Several options as listed in Tab. 30 can be selected to optimize the split performance.

Table 30: Split techniques for zonotopes.

| Split technique | Comment | Reference |
|---|---|---|
| `splitOneGen` | splits one generator | [33, Proposition 3.8] |
| `directionSplit` | splits all generators in one direction | — |
| `directionSplitBundle` | exact split using zonotope bundles | [42, Section V.A] |
| `halfspaceSplit` | split along a given halfspace | — |

### A.1.2 Method `norm`

This function can compute the Euclidean norm of the zonotope vertex with the biggest Euclidean distance from the center (without enumerating vertices). Although this problem has exponential

worst-case complexity in the number of generators, by using a more advanced branch-and-bound solver like Gurobi[33] with YALMIP, the computation time can be reduced significantly.

### A.1.3 Method `ellipsoid`

Tab. 31 shows available conversions from a zonotope $Z$ to an ellipsoid $E$. Results specified (o: overapproximation, u: underapproximation)

- by `o:exact`, `u:exact` are the optimal minimum-volume enclosing and maximum-volume inscribed ellipsoids. For more detail, see [125, Sec. 8.4.1, Sec. 8.4.2].
- by `o:norm`,`u:norm` approximate `o:exact`,`u:exact` by using the exact zonotope norm.
- by `o:norm:bnd`, `u:norm:bnd` are the same as their respective `*:norm` specifier, but use a tractably computable bound on the zonotope norm.

Table 31: Available zonotope → ellipsoid conversions with "+", "−" meaning polynomial and exponential complexity with respect to generator count, respectively.

| Specifier | Mode | Complexity |
|---|---|---|
| `o:exact` | o | − |
| `o:norm` | o | − |
| `o:norm:bnd` | o | + |
| `u:exact` | u | − |
| `u:norm` | u | − |
| `u:norm:bnd` | u | + |

## A.2 Intervals

In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5) the class `interval` supports additional mehtods. Since the `interval` class has a lot of methods, we separate them into methods that realize mathematical functions and methods that do not realize mathematical functions.

**Methods realizing mathematical functions and operations**

- `abs` – returns the absolute value as defined in [11, Eq. (10)].
- `acos` – arccos(·) function as defined in [11, Eq. (6)].
- `acosh` – arccosh(·) function as defined in [11, Eq. (8)].
- `asin` – arcsin(·) function as defined in [11, Eq. (6)].
- `asinh` – arcsinh(·) function as defined in [11, Eq. (8)].
- `atan` – arctan(·) function as defined in [11, Eq. (6)].
- `atanh` – arctanh(·) function as defined in [11, Eq. (8)].
- `cos` – cos(·) function as defined in [11, Eq. (13)].
- `cosh` – cosh(·) function as defined in [11, Eq. (7)].
- `ctranspose` – overloaded ''' operator for single operand to transpose a matrix.

[33] https://www.gurobi.com/

- `enlarge` – enlarges each dimension by a factor around its mean value.
- `eq` – overloads the '==' operator to check if both intervals are equal.
- `exp` – exponential function as defined in [11, Eq. (4)].
- `horzcat` – overloads horizontal concatenation.
- `infimum` – returns the infimum.
- `isscalar` – returns true if the interval is one-dimensional, false otherwise.
- `le` – overloads `<=` operator: Is one interval equal or the subset of another interval?
- `log` – natural logarithm function as defined in [11, Eq. (5)].
- `lt` – overloads `<` operator: Is one interval equal or the subset of another interval?
- `minus` – overloaded '-' operator, see [11, Eq. (2)].
- `mpower` – overloaded '^' operator (power), see [11, Eq. (9)].
- `mrdivide` – overloaded '/' operator (division), see [11, Eq. (3)].
- `mtimes` – overloaded '*' operator (multiplication), see [11, Eq. (2)] for scalars and [11, Eq. (16)] for matrices.
- `ne` – overloaded ' =' operator.
- `power` – overloaded '.^' operator for intervals (power), see [11, Eq. (9)].
- `prod` – product of array elements.
- `rdivide` – overloads the './' operator: provides elementwise division of two matrices.
- `sin` – $\sin(\cdot)$ function as defined in [11, Eq. (12)].
- `sinh` – $\sinh(\cdot)$ function as defined in [11, Eq. (7)].
- `sqrt` – $\sqrt{(\cdot)}$ function as defined in [11, Eq. (5)].
- `tan` – $\tan(\cdot)$ function as defined in [11, Eq. (14)].
- `tanh` – $\tanh(\cdot)$ function as defined in [11, Eq. (7)].
- `times` – overloaded '.*' operator for elementwise multiplication of matrices.
- `transpose` – overloads the ' .' ' operator to compute the transpose of an interval matrix.
- `uminus` – overloaded '-' operator for a single operand.
- `uplus` – overloaded '+' operator for single operand.

**Other methods**

- `diag` – create diagonal matrix or get diagonal elements of matrix.
- `enlarge` – enlarges an `interval` object around its center.
- `gridPoints` – computes grid points of an interval; the points are generated in a way such that a continuous space is uniformly partitioned.
- `horzcat` – overloads the operator for horizontal concatenation, e.g., `a = [b,c,d]`.
- `infimum` – returns the infimum of an interval.
- `isscalar` – returns 1 if interval is scalar and 0 otherwise.

- `length` – overloads the operator that returns the length of the longest array dimension.
- `partition` – partitions a multidimensional interval into subintervals.
- `rad` – returns the radius (= 0.5·width) of an interval.
- `radius` – computes the radius of a hypersphere enclosing an interval.
- `reshape` – overloads the operator 'reshape' for reshaping matrices.
- `size` – overloads the operator that returns the size of the object, i.e., length of an array in each dimension.
- `split` – splits an interval in one dimension.
- `subsasgn` – overloads the operator that assigns elements of an interval matrix `I`, e.g., `I(1,2)=value`, where the element of the first row and second column is set.
- `subsref` – overloads the operator that selects elements of an interval matrix `I`, e.g., `value=I(1,2)`, where the element of the first row and second column is read.
- `sum` – overloaded 'sum()' operator for intervals.
- `supremum` – returns the supremum of an interval.
- `vertcat` – overloads the operator for vertical concatenation, e.g., `a = [b;c;d]`.

## A.3 Ellipsoids

In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5) the class `ellipsoid` supports the following methods:

- `distance` – computes the smallest euclidean distance between an ellipsoid and another set representation.
- `enlarge` – enlarges the ellipsoid by a scalar factor for each dimension.
- `eq` – overloads the '==' operator to check if two ellipsoids are equal.
- `intersectStrip` – computes the intersection of a ellipsoid and a list of strips.
- `minus` – computes the Minkowski difference as defined in [35].
- `radius` – returns the radius of the smallest hyper-sphere which contains a given ellipsoid
- `rank` – returns the rank of an ellipsoid

### A.3.1 Method `plus`

In [35, Sec. 2.2.2], an approach to compute an ellipsoidal overapproximation of the Minkowski sum of two $n$-dimensional ellipsoids is provided. The method `plus` (overwriting `+`) uses this approach to compute the resulting ellipsoid for $2n$ roughly uniformly sampled unit directions, intersect all of the resulting ellipsoids and calculate the intersection overapproximation using `and`.

However, since in many cases, one can often find a direction such that the result in that direction has a smaller volume than on average, we also support an overloaded method `and(E1,E2,L)` which allows to specify custom direction(s).

### A.3.2 Method `zonotope`

In [126], inner and outer approximations for both ellipsoids and zonotopes are presented. Tab. 32 shows these conversions from an ellipsoid $E$ to a zonotope $Z$ where $m$ is the user-specified number of generators. Results specified (o: outer approximation, i: inner approximation)

- by `o:box`,`u:box` are the optimal minimum-volume and maximum-volume parallelotopes,
- by `o:norm`,`u:norm` approximate $Z$ arbitrarily closely for arbitrary $m$ using the exact zonotope norm,
- by `o:norm:bnd`, `u:norm:bnd` are the same as their respective `*:norm` specifier, but use a tractably computable bound on the zonotope norm.

Table 32: Available ellipsoid → zonotope conversions with "+", "−" meaning polynomial and exponential complexity with respect to dimension, respectively.

| Specifier | Mode | Complexity |
|---|---|---|
| `o:box` | o | + |
| `o:norm` | o | − |
| `o:norm:bnd` | o | + |
| `i:box` | i | + |
| `i:norm` | i | − |
| `i:norm:bnd` | i | + |

### A.3.3 Method `distance`

Computes the euclidean distance between the second argument `S` and the ellipsoid `E`, where `distance(E,S)`$> 0$ means the two objects do not intersect. For `distance(E,S)`$= 0$, `S` and `E` either touch or intersect. For `S` being a hyperplane, `distance(E,S)`$= 0$ means `S` and `E` touch, and `distance(E,S)`$< 0$ represents a real intersection.

## A.4 Polytopes

In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5), the class `polytope` supports the following methods:

- `eq` – overloads the '==' operator to check if two polytopes are equal.
- `eventFcn` – event function that detects is a trajectory enters the set. This function is required for the simulation of hybrid systems (see Appendix C).
- `constraints` – computes the halfspace representation of the polytope.
- `hausdorffDist` – alculates the Hausdorff distance between a polytope and a set or a point
- `le` – overloads the '<=' operator; returns 1 if one polytopes is equal or enclosed by the other one and 0 otherwise.
- `minus` – overloaded '-' operator for the subtraction of a vector from an `polytope` or the Minkowski difference between two `polytope` objects.
- `mldivide` – computes the set difference $P_1 \setminus P_2$ such that $P_2$ is subtracted from $P_1$.
- `projectHighDim` – projects a polytope to a higher-dimensional space.

## A.5 Polynomial Zonotopes

In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5), the class `polyZonotope` supports the following methods:

- `approxVolumeRatio` – computes the approximate ratio of the volumes between the dependent generator and the independent generator part of the polynomial zonotope.
- `compact` – removes redundancies in the representation of the polynomial zonotope.
- `containsPointSet` – checks if a point set is fully enclosed by a tight over-approximation of a polynomial zonotope.
- `deleteZeros` – deletes all generators of length 0.
- `exactPlus` – compute the addition of two sets while preserving the dependencies between the two sets.
- `fhandle` – computes a function handle based on the given polynomial zonotope.
- `getSubset` – extracts a subset by specifying new ranges for the factors.
- `hausdorffDist` – calculates an approximation of the Hausdorff distance between a polynomial zonotope and a point cloud.
- `hessianHandle` – computes a function handle which returns the hessian matrix at the given point.
- `innerApprox` – returns an inner-approximation of a polynomial zonotope with a union of zonotopes.
- `isInterval` – checks if a polynomial zonotope represents an interval.
- `isPolytope` – checks if a polynomial zonotope represents a polytope.
- `isZero` – check for each dimension if polynomial zonotope is equal to zero.
- `isZonotope` – checks if a polynomial zonotope represents a zonotope.
- `jacobian` – computes the derivatives of a given polynomial zonotope.
- `jacobianHandle` – computes a function handle which calculates the jacobian matrix at a given point.
- `noIndep` – remove the independent generators from a polynomial zonotope.
- `onlyId` – returns a polynomial zonotope with only specified ids as well as the remaining polynomial zonotope.
- `partZonotope` – computes a zonotope over-approximation in the specified id entries only.
- `plotRandPoint` – plots a point cloud of random points inside a polynomial zonotope.
- `polygon` – creates a polygon enclosure of a two-dimensional polynomial zonotope.
- `replaceId` – replaces specified id entries with others.
- `resolve` – replaces specified id entries with given numerical values for corresponding dependent factors.
- `restoreId` – adds (if not alreay there) specified ids

- `restructure` – Calculate a new over-approxmiating representation of a polynomial zonotope in such a way that there remain no independent generators. More information can be found in Sec. 7.4.
- `split` – splits a polynomial zonotope into two or more polynomial zonotopes that enclose the original polynomial zonotope.
- `splitDepFactor` – splits one dependent factor of a polynomial zonotope.
- `splitLongestGen` – splits the longest generator dependent generator with a polynomial order of 1 for a polynomial zonotope.
- `stack` – extends dimensionality with provided polynomial zonotopes while preserving dependencies.
- `subs` – computes the functional composition of two polynomial zonotopes.
- `sum` – computes the sum of multiple polynomial zonotopes.

### A.5.1 Method `jacobian`

For a $n$-dimensional polynomial zonotope `pZ` with $N$ dependent factors, `jacobian(pZ)` returns a $N$-dimensional cell array where each element is the respective derivative of `pZ` with dimension $n$.

### A.5.2 Method `jacobianHandle`

For a $n$-dimensional polynomial zonotope `pZ`, where `id` contains some (in any order) or all ids of `pZ`, `jacobianHandle(pZ,id)` returns a function handle of the form `@(x,p)H(x,p)`. This handle returns the corresponding jacobian matrix of size $n$ by `numel(id)` at `x`, `p`, where `x` are treated as variables, and `p` as parameters.

Accepts symbolic vectors for `x` and `p`.

### A.5.3 Method `hessianHandle`

For a 1D polynomial zonotope `pZ`, where `id` contains some (in any order) or all ids of `pZ`, `hessianHandle(pZ,id)` returns a function handle of the form `@(x,p)H(x,p)`. This handle returns the corresponding square, symmetric hessian matrix of size `numel(id)` by `numel(id)` at `x` and `p`, where `x` are treated as variables, and `p` as parameters.

Accepts symbolic vectors for `x` and `p`.

## A.6 Capsule

In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5), the class `capsule` supports the following methods:

- `enlarge` – enlarges the capsule around its center.
- `polygon` – under-approximates a two-dimensional capsule by a polygon and returns its vertices. This function is mainly used for plotting.
- `radius` – returns the radius of the enclosing hyperball.

## A.7 Zonotope Bundles

In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5), the class `zonoBundle` supports the following methods:

- `encloseTight` – generates a zonotope bundle that encloses two zonotopes bundles in a possibly tighter way than `enclose` as outlined in [42, Sec. VI.A].
- `enlarge` – enlarges the generators of each zonotope in the bundle by a vector of factors for each dimension.
- `reduceCombined` – reduces the order of a zonotope bundle by not reducing each zonotope separately as in `reduce`, but in a combined fashion.
- `replace` – replaces a zonotope at an index position by another zonotope.
- `shrink` – shrinks the size of individual zonotopes by explicitly computing the intersection of individual zonotopes; however, in total, the size of the zonotope bundle will increase. This step is important when individual zonotopes are large, but the zonotope bundles represents a small set. In this setting, the over-approximations of some operations, such as `mtimes` might become too over-approximative. Although `shrink` initially increases the size of the zonotope bundle, subsequent operations are less over-approximative since the individual zonotopes have been shrunk.
- `split` – splits a zonotope bundle into two or more zonotopes bundles. Other than for zonotopes, the split is exact. The method can split halfway in a particular direction or given a separating hyperplane.

## A.8 Constrained Zonotopes

In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5), the class `conZonotope` supports the following methods:

- `conIntersect` – add the constraint that the linear transformation of a constrained zonotope intersects another constrained zonotope
- `deleteZeros` – deletes generators whose entries are all zero.
- `intersectStrip` – computes the intersection of a constrained zonotope and a list of strips.
- `intervalMultiplication` – computes the multiplication of an interval with a constrained zonotope, this function is called by the function `mtimes`.
- `minus` – computes the Minkowski difference of two constrained zonotopes.
- `plotZono` – plots a two-dimensional projection of the `conZonotope` object together with the corresponding zonotope.
- `reduceConstraints` – reduces the number of constraints of a constrained zonotope.
- `rescale` – prune the domain of the zonotope factors $\beta_i$ by adequate adaption of the zonotope generators. More details can be found in [34].
- `split` – splits a constrained zonotope into two or more constrained zonotopes that enclose the original constrained zonotope.

### A.8.1 Method `reduce`

One parameter to describe the complexity of a constrained zonotope is the *degrees-of-freedom order* $o_c = (p-q)/n$, where $p$ represents the number of generators, $q$ is the number of constraints and $n$ is the state space dimension. The method `reduce` implements the two options reduction of the number of constraints $q$ [34, Section 4.2] and reduction of the *degrees-of-freedom order* $o_c$ [34, Section 4.3].

## A.9 Probabilistic Zonotopes

In addition to the standard set operations described in Sec. 2.1 and the methods for converting between set operations (see Tab. 5), the class `probZonotope` supports the following methods:

- `abs` – returns a probabilistic zonotope with absolute values of the center and the interval generator vectors.
- `enclosingPolytope` – converts the mean of a probabilistic zonotope to a polytope representation.
- `enclosingProbability` – computes values to plot the mesh of a two-dimensional projection of the enclosing probability hull.
- `generators` – returns the generator matrix of a probabilistic zonotope using its covariance matrix Sigma.
- `max` – computes an over-approximation of the maximum on the m-sigma bound according to [44, Equation 3].
- `mean` – returns the uncertain mean of a probabilistic zonotope.
- `probReduce` – reduces the number of single Gaussian distributions to the dimension of the state space.
- `pyramid` – encloses a probabilistic zonotope $\mathscr{Z}$ by a pyramid with step sizes defined by an array of confidence bounds and determines the probability of intersection with a polytope $\mathcal{P}$ as described in [44, Section VI.C].
- `reduce` – returns an over-approximating zonotope with fewer generators. The zonotope of the uncertain mean $\mathcal{Z}$ is reduced, while the order reduction of the probabilistic part is done by the method `probReduce`.
- `sigma` – returns the $\Sigma$ matrix of a probabilistic zonotope.
- `singleGenPlot` – plots a two-dimensional projection of a probabilistic zonotope with a maximum of 5 generators.
- `sup` – returns the supremum by $||\cdot||_\infty$ of the probabilistic zonotope.

## A.10 Level Sets

In addition to the methods described in Sec. 2.1, we support the following methods for the class `levelSet`:

- `eventFcn` – event function that detects is a trajectory enters the set. This function is required for the simulation of hybrid systems (see Appendix C).
- `tightenDomain` – contracts the interval domain for the intersection between a level set and another set.

### A.10.1 Method `plot`

Since level sets can in general be unbounded, it is often impossible to plot the whole level set. When plotting a level set we therefore first extract the area of the state space that is shown in the current plot, and then plot the intersection between this area and the level set. Consequently, it is important to first define the desired area of the plot using MATLABs `xlim` and `ylim` functions before the level set is plotted.

Furthermore, the projection of a level set on two dimensions usually fills the whole space and is therefore not very interesting. Instead of plotting the real projection, we set all states that do not belong to the current projection equal to 0 and then plot the resulting set which is a level set in 2D.

## A.11 Taylor Models

Since this class has a lot of methods, we separate them into methods that realize mathematical functions and methods that do not realize mathematical functions.

### Methods realizing mathematical functions and operations

- `acos` – arccos(·) function as defined in [12, Eq. (31)].
- `asin` – arcsin(·) function as defined in [12, Eq. (30)].
- `atan` – arctan(·) function as defined in [12, Eq. (32)].
- `cos` – cos(·) function as defined in [12, Eq. (25)].
- `cosh` – cosh(·)function as defined in [12, Eq. (28)].
- `det` – determinant of a Taylor model matrix.
- `exp` – exponential function as defined in [12, Eq. (21)].
- `log` – natural logarithm function as defined in [12, Eq. (22)].
- `minus` – overloaded '-' operator, see [12, Eq. (7)].
- `mpower` – overloaded '^' operator (power).
- `mrdivide` – overloaded '/' operator (division), see [12, Eq. (9)].
- `power` – overloaded '.^' operator (elementwise power).
- `rdivide` – overloads the './' operator: provides elementwise division of two matrices.
- `reexpand` – re-expand the Taylor model at a new expansion point.
- `sin` – sin(·) function as defined in [12, Eq. (24)].
- `sinh` – sinh(·) function as defined in [12, Eq. (27)].
- `sqrt` – $\sqrt{(\cdot)}$ function as defined in [12, Eq. (23)].
- `tan` – tan(·) function as defined in [12, Eq. (26)].
- `tanh` – tanh(·) function as defined in [12, Eq. (29)].
- `times` – overloaded '.*' operator for elementwise multiplication of matrices.
- `trace` – trace of a Taylor model matrix.
- `uminus` – overloaded '-' operator for a single operand.
- `uplus` – overloaded '+' operator for a single operand.

### Other methods

- `getCoef` – returns the array of polynomial coefficients of a `taylm` object.
- `getRem` – returns the interval part of a `taylm` object.

- `getSyms` – returns the polynomial part of a `taylm` object as a symbolic expression.
- `optBernstein` – range bounding using Bernstein polynomials.
- `optBnb` – implementation of the branch and bound algorithm as presented in [12, Sec. 2.3.2].
- `optBnbAdv` – implementation of the advanced branch and bound algorithm as presented in [12, Sec. 2.3.2].
- `optLinQuad` – implementation of the algorithm based on LDB and QFB as presented in [12, Sec. 2.3.3].
- `prod` – product of array elements.
- `set` – set the additional class parameters (see [12, Sec. 4.3]).
- `setName` – set the names of the variables in `taylm`.
- `subsasgn` – overloads the operator that assigns elements of a `taylm` matrix `I`, e.g., `I(1,2) = value`, where the element of the first row and second column is set.
- `subsref` – overloads the operator that selects elements of a `taylm` matrix `I`, e.g., `value = I(1,2)`, where the element of the first row and second column is read.

### A.11.1 Creating Taylor Models

Here we describe the different ways to create an object of class `taylm`. To make use of cancellation effects, we have to provide names for variables in order to recognize identical variables; this is different from implementations of interval arithmetic, where each variable is treated individually. We have realized three primal ways to generate a matrix containing Taylor models.

**Method 1: Composition from scalar Taylor models.**

The first possibility is to generate scalar Taylor models from intervals as shown subsequently.

```
1 a1 = interval(-1, 2); % generate a scalar interval [-1,2]
2 b1 = taylm(a1, 6); % generate a scalar Taylor model of order 6
3 a2 = interval(2, 3); % generate a scalar interval [2,3]
4 b2 = taylm(a2, 6); % generate a scalar Taylor model of order 6
5 c = [b1; b2] % generate a row of Taylor models
```

When a scalar Taylor model is generated from a scalar interval, the name of the variable is deduced from the name of the interval. If one wishes to overwrite the name of a variable `a2` to `c`, one can use the command `taylm(a2, 6, {'c'})`.

**Method 2: Converting an interval matrix.**

One can also first generate an interval matrix, i.e., a matrix containing intervals, and then convert the interval matrix into a Taylor model. The subsequent example generates the same Taylor model as in the previous example.

```
1 a = interval([-1;2], [2;3]); % generate an interval vector [[-1,2]; [2,3]]
2 c = taylm(a, 6, {'a1';'a2'}) % generate Taylor model (order 6)
```

Note that the cell for naming variables `{'a1';'a2'}` has to have the same dimensions as the interval matrix `a`. If no names are provided, default names are automatically generated.

**Method 3: Symbolic expressions.**

We also provide the possibility to create a Taylor model from a symbolic expression.

```
syms a1 a2; % instantiate symbolic variables
s = [2 + 1.5*a1; 2.75 + 0.25*a2]; % create symbolic function
c = taylm(s, interval([-2;-3],[0;1]), 6)  % generate Taylor model
```

This method does not require naming variables since variable names are taken from the variable names of the symbolic expression. The interval of possible values has to be specified after the symbolic expression $s$; here: $[[-2,0]\,[-3,1]]^T$.

All examples generate a row vector $c$. Since all variables are normalized to the range $[-1,1]$, we obtain

$$c = \begin{bmatrix} 0.5 + 1.5 \cdot \tilde{a}_1 + [0,0] \\ 2.5 + 0.5 \cdot \tilde{a}_2 + [0,0] \end{bmatrix} .$$

The following workspace output of MATLAB demonstrates how the dependency problem is considered by keeping track of all encountered variables:

```
>> c(1) + c(1)
ans =
        1.0 + 3.0*a1 + [0.00000,0.00000]

>> c(1) + c(2)
ans =
        3.0 + 1.5*a1 + 0.5*a2 + [0.00000,0.00000]
```

### A.12 Deprecated Functionality

In a continuous effort to harmonize functionality across CORA and improve the user experience, some older functionality gets deprecated with each release. Tab. 33 lists all currently deprecated functionality along with their replacement. A warning is printed into the command window if a user calls a deprecated function. We encourage users to switch to the suggested replacement as soon as possible as continuing using the deprecated functionality might result in unexpected behavior and eventually broken code. Former functionality which is no longer supported is listed in Tab. 34.

## B Additional Methods for Matrix Set Representations

In addition to the set operations described in Sec. 3.1 and the methods for converting between set operations (see Tab. 5), all matrix set representations implement additional methods, which are documented subsequently.

### B.1 Matrix Polytopes

We support the following additional methods for matrix polytopes:

- `expmInd` – operator for the exponential matrix of a matrix polytope, evaluated independently.
- `expmIndMixed` – operator for the exponential matrix of a matrix polytope, evaluated independently. Higher order terms are computed via interval arithmetic.
- `mpower` – overloaded '^' operator for the power of matrix polytopes.

Table 33: List of currently deprecated functionality along with their replacement.

| Functionality | Replacement | Since | Description |
|---|---|---|---|
| `contDynamics.dim` | `contDynamics.nrOfStates` | v2025 | This change was made to be consistent with the other properties. |
| `polygon.simplify` | `polygon.compact` | v2025 | This change was made in an effort to unify the syntax across all set representations. |
| `conHyperplane` | `polytope` | v2025 | Constrained hyperplanes are a special case of polytopes. As the benefit of having an additional class for this special case is minor, we removed it to improve maintainability. |
| `halfspace` | `polytope` | v2025 | Halfspaces are a special case of polytopes. As the benefit of having an additional class for this special case is minor, we removed it to improve maintainability. |
| `reset` struct | `non\|linearReset` | v2025 | This change was made to improve code reliability. |
| `matPolytope/constructor` | see description | v2024.2.0 | To specify the vertices, use a single numeric matrix with dimensions (n x m x N) instead of a cell array with N vertices each with dimensions (n x m). |
| `matZonotope/constructor` | see description | v2024.2.0 | To specify the generators, use a single numeric matrix with dimensions (n x m x N) instead of a cell array with N generators each with dimensions (n x m). |
| `conPolyZono.Grest` | `conPolyZono.GI` | v2024 | This change was made to be consistent with the notation in papers. |
| `conPolyZono.expMat` | `conPolyZono.E` | v2024 | This change was made to be consistent with the notation in papers. |
| `conPolyZono.expMat_` | `conPolyZono.EC` | v2024 | This change was made to be consistent with the notation in papers. |
| `conZonotope.Z` | `conZonotope.c/G` | v2024 | This change was made to be consistent with the notation in papers. |
| `conZonotope/deleteZeros` | `compact(cZ,'zeros')` | v2024 | This change was made in an effort to unify the syntax across all set representations. |
| `contSet/is<Name>` | `representsa(S,'<name>')` | v2024 | This change was made in an effort to unify the syntax across all set representations. |
| `contSet/isZero` | `representsa(S,'origin')` | v2024 | This change was made in an effort to unify the syntax across all set representations. |
| `contSet/isEmpty` | `representsa(S,'emptySet')` | v2024 | This change was made in an effort to unify the syntax across all set representations. Additionally, the function `isempty` is also called implicitly by MATLAB in various circumstances, e.g., when shown in the workspace, which might lead to long loading times. |
| `polyZonotope.Grest` | `polyZonotope.GI` | v2024 | This change was made to be consistent with the notation in papers. |
| `polyZonotope.expMat` | `polyZonotope.E` | v2024 | This change was made to be consistent with the notation in papers. |
| `zonotope.Z` | `zonotope.c/G` | v2024 | This change was made to be consistent with the notation in papers. |
| `zonotope.halfspaces` | `zonotope/polytope` | v2024 | This change was made to avoid code redundancy. |

Table 34: List of currently deprecated functionality along with their replacement.

| Functionality | Replacement | Since | Reason |
|---|---|---|---|
| `contSet/in` | `contSet/contains` | v2022 | The main reason is that the syntax `in(S1,S2)` can also be written as `S1.in(S2)`. This is, however, the opposite of the actual computation, which checks whether `S2` is a subset of `S1`. The function `contains` eliminates this potential confusion, as `S1.contains(S2)` is semantically correct. |

- `plot` – plots 2-dimensional projection of a matrix polytope.
- `simplePlus` – computes the Minkowski addition of two matrix polytopes without reducing the vertices by a convex hull computation.

## B.2 Matrix Zonotopes

We support the following additional methods for matrix zonotopes:

- `concatenate` – concatenates the center and all generators of two matrix zonotopes.
- `dependentTerms` – considers dependency in the computation of Taylor terms for the matrix zonotope exponential according to [118, Proposition 4.3].
- `expmInd` – operator for the exponential matrix of a matrix zonotope, evaluated independently.
- `expmIndMixed` – operator for the exponential matrix of a matrix zonotope, evaluated independently. Higher order terms are computed via interval arithmetic.
- `expmMixed` – operator for the exponential matrix of a matrix zonotope, evaluated dependently. Higher order terms are computed via interval arithmetic as discussed in [118, Section 4.4.4].
- `expmOneParam` – operator for the exponential matrix of a matrix zonotope when only one parameter is uncertain as described in [60, Theorem 1].
- `mpower` – overloaded '^' operator for the power of matrix zonotopes.
- `norm` – computes exactly the maximum norm value of all possible matrices.
- `plot` – plots 2-dimensional projection of a matrix zonotope.
- `powers` – computes the powers of a matrix zonotope up to a certain order.
- `randomSampling` – creates random samples within a matrix zonotope.
- `reduce` – reduces the order of a matrix zonotope. This is done by converting the matrix zonotope to a zonotope, reducing the zonotope, and converting the result back to a matrix zonotope.
- `subsref` – overloads the operator that selects elements of a `matZonotope`.
- `volume` – computes the volume of a matrix zonotope by computing the volume of the corresponding zonotope.
- `zonotope` – converts a matrix zonotope into a zonotope.

### B.3 Interval Matrices

We support the following additional methods for interval matrices:

- `abs` – returns the absolute value bound of an interval matrix.
- `dependentTerms` – considers dependency in the computation of Taylor terms for the interval matrix exponential according to [118, Proposition 4.4].
- `exactSquare` – computes the exact square of an interval matrix.
- `expmAbsoluteBound` – returns the over-approximation of the absolute bound of the symmetric solution of the computation of the exponential matrix.
- `expmInd` – operator for the exponential matrix of an interval matrix, evaluated independently.
- `expmIndMixed` – dummy function for interval matrices.
- `expmMixed` – dummy function for interval matrices.
- `expmNormInf` – returns the over-approximation of the norm of the difference between the interval matrix exponential and the exponential from the center matrix according to [118, Theorem 4.2].
- `exponentialRemainder` – returns the remainder of the exponential matrix according to [118, Proposition 4.1].
- `interval` – converts an interval matrix to an interval.
- `mpower` – overloaded '^' operator for the power of interval matrices.
- `mtimes` – standard method, see Sec. 3.1.1 for numeric matrix multiplication or a multiplication with another interval matrix according to [118, Equation 4.11].
- `norm` – computes exactly the maximum norm value of all possible matrices.
- `plot` – plots 2-dimensional projection of an interval matrix.
- `powers` – computes the powers of an interval matrix up to a certain order.
- `randomIntervalMatrix` – generates a random interval matrix with a specified center and a specified delta matrix or scalar. The number of elements of that matrix which are uncertain has to be specified, too.
- `randomSampling` – creates random samples within a matrix zonotope.
- `subsref` – overloads the operator that selects elements.
- `volume` – computes the volume of an interval matrix by computing the volume of the corresponding interval.

## C Simulation of Hybrid Automata

While the reachable set computation of hybrid systems as performed in CORA is described in several publications, see e.g., [33, 99, 101], the simulation of hybrid systems is nowhere documented. For this reason, the simulation is described in this subsection in more detail. The simulation is performed by applying the following steps:

① Preparation 1: Guard sets and invariants can be specified by any set representation that CORA offers. For simulation purposes, all set representations are transformed into a halfspace representation as illustrated in Fig. 3(b). This is performed by transforming

intervals, zonotopes, and zonotope bundles to a polytope, see Tab. 5. Next, of all polytopes the halfspace generation is obtained. Guards that are already defined as halfspaces do not have to be converted, of course. In the end, one obtains a set of halfspaces for guard sets and the invariant for each location. The result for one location is shown in Fig. 54.

② Preparation 2: The ordinary differential equation (ODE) solvers of MATLAB can be connected to so-called *event functions*. If during the simulation, one of the event functions has a zero crossing, MATLAB stops the simulation and goes forward and backward in time in an iterative way to determine the zero crossing up to some numerical precision. It can be set if the ODE solver should react to a zero crossing when the event function changes from negative to positive (`direction=+1`), the other way round (`direction=-1`), or in any direction (`direction=0`). It can also be set if the simulation should stop after a zero crossing or not.

CORA automatically generates an event function for each halfspace, where the simulation is stopped when the halfspace of the invariant is left (`direction=+1`) and stopped for halfspaces of guard sets when the halfspace is entered (`direction=-1`). In any case, the simulation will stop.

③ During the simulation, the integration of the ODE stops as soon as any event function is triggered. This, however, does not necessarily mean that a guard set is hit as shown in Fig. 54(b). Only when the state is on the edge of a guard set, the integration is stopped for the current location. Otherwise, the integration is continued. Please note that it is not sufficient to check whether a state during the simulation enters a guard set, since this could cause missing a guard set as shown in Fig. 55.

④ After a guard set is hit, the discrete state changes according to the transition function and the continuous state according to the jump function as described above. Currently, only urgent semantics is implemented in CORA, i.e., a transition is taken as soon as a guard set is hit, although the guard might model non-deterministic switching. The simulation continues with step ③ in the next location until the time horizon is reached.

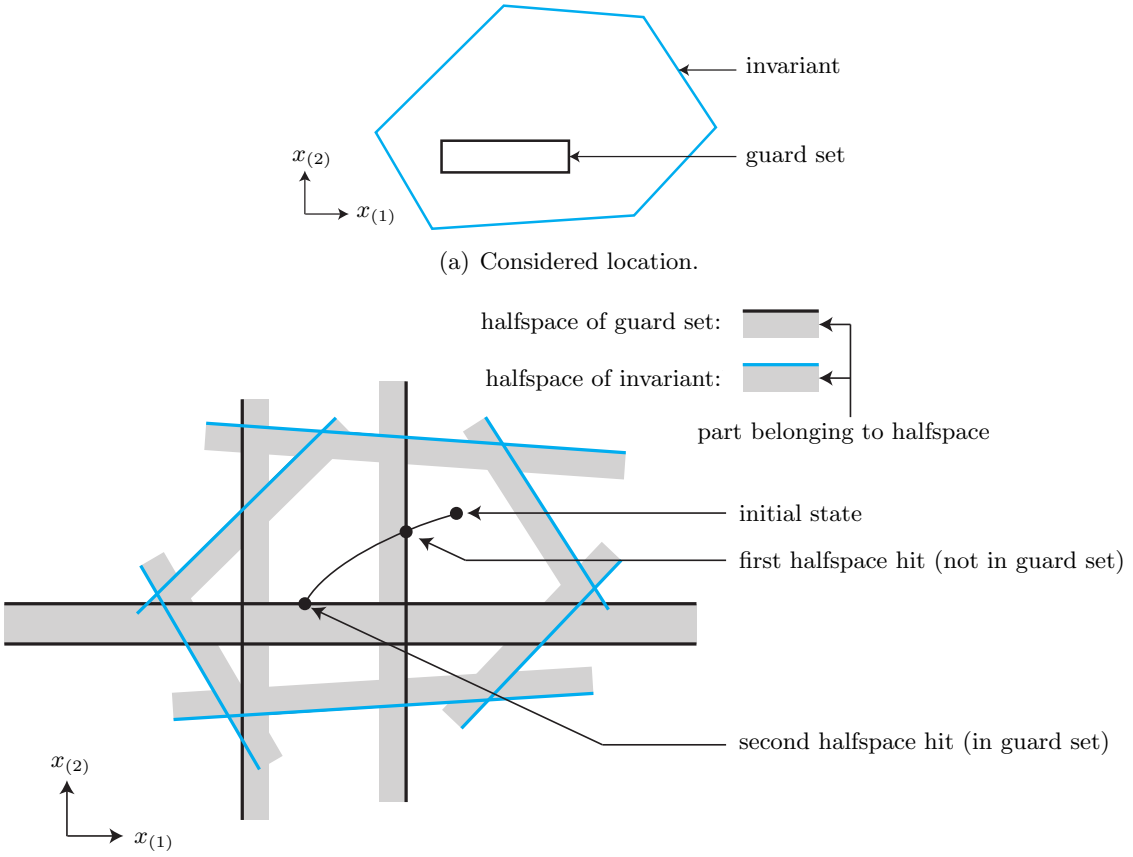
# D Implementation of Loading SpaceEx Models

This section describes the implementation details of the spaceex2cora converter. We will first briefly describe the SpaceEx format in Appendix D.1, followed by an overview of the conversion in Appendix D.2. Details of the conversion are presented in Appendices D.3 and D.4.

## D.1 The SpaceEx Format

The SpaceEx format [117] has similarities to statecharts [127]. A SpaceEx model is composed of network and base components. Base components resemble XOR states in statecharts, which in essence describe a monolithic hybrid automaton (see Sec. 4.3) of which not all components have to be specified, e.g., one does not have to specify a flow function if a base component is a static controller. Analogously to XOR states, only one base component can be active at the same time. Network components resemble AND states of statecharts and bind base components. As in AND states of statecharts, several base components can be active at the same time. SpaceEx models can be seen as a tree of components, where base components are the leaves and the root of the tree defines the interface (i.e., states & inputs) of the complete model consisting of all components.

When a component is bound by a network component, all variables of the bound component (states, inputs, constant parameters) must be mapped to variables of the binding component or to numerical values. If a component is bound multiple times, each bind creates a new instance




(a) Considered location.

(b) Simulation using halfspaces.

Figure 54: Illustration of the algorithm for simulating a hybrid automaton.

of that component with independent variables. This makes it convenient to reuse existing model structures, e.g., when one requires several heaters in a building, but the dynamics of each heater has the same structure but different parameters.

The SpaceEx modeling language is described in greater detail on the SpaceEx website[34].

## D.2 Overview of the Conversion

The conversion of SpaceEx models to CORA models is achieved in two phases. In the first phase, the XML structure is parsed and a MATLAB struct of the model is generated. This is realized in the converter function `spaceex2cora.m` when it calls

```
structHA = SX2structHA('model.xml','mainComponent')
```

returning the MATLAB structure `structHA`. The optional second argument specifies the *highest-ranking network component*, from which the model is loaded. In XML files containing just one model that is always the last defined component (default component). Please note that the function `SX2structHA` has verbose output. Please check any warnings issued, as they might indicate an incomplete conversion. For details see the restrictions mentioned in Sec. 8.2.

In the second phase, the computed `structHA` is used to create a MATLAB function that when executed instantiates the CORA model. This MATLAB function is created by

[34] http://spaceex.imag.fr/sites/default/files/spaceex_modeling_language_0.pdf

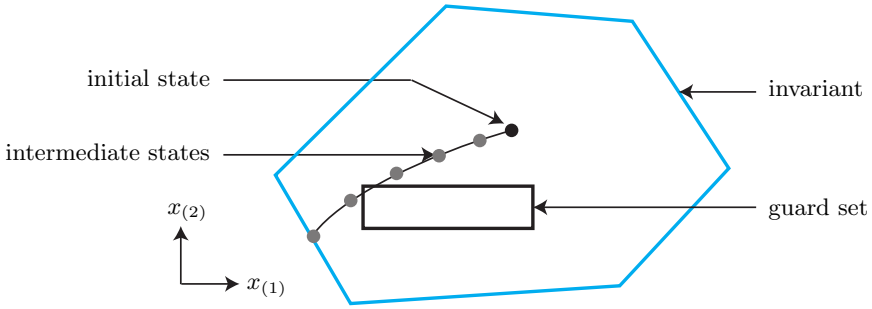



Figure 55: Guard intersections can be missed when one only checks whether intermediate states are in any guard set.

```
StructHA2file(structHA,'myModel','my/cora/files').
```

Calling `myModel()` instantiates the CORA model converted from the original SpaceEx model; this is demonstrated for a bouncing ball example in Sec. 8.2.

## D.3 Parsing the SpaceEx Components (Phase 1)

Parsing the SpaceEx components is performed in five steps:

1. Accessing XML files (Appendix D.3.1);
2. Parsing component templates (Appendix D.3.2);
3. Building component instances (Appendix D.3.3);
4. Merging component instances (Appendix D.3.4);
5. Conversion to state-space form (Appendix D.3.5).

These steps are described in detail subsequently.

### D.3.1 Accessing XML Files

We use the popular function `xml2struct` (Falkena, Wanner, Smirnov) from the MATLAB File Exchange to conveniently analyze XML files. The function converts XML structures such as

```
<mynode id="1" note="foobar">
        <foo>FOO</foo>
        <bar>BAR</bar>
</mynode>
```

to a nested MATLAB struct:

*MATLAB struct*
- mynode
  - Attributes
    - id: *'1'*
    - description: *'foobar'*
  - foo
    - Text: *'FOO'*
  - bar
    - Text: *'BAR'*

The resulting MATLAB struct realizes an intuitive access to attributes and an easy extraction of sub-nodes in MATLAB.

### D.3.2 Parsing Component Templates

Before we begin with the semantic evaluation, base components and network components are parsed into a more convenient format.

#### Base components

For base components we convert equations stored as strings specifying flow, invariants, guards, and resets, to a more compact and manipulatable format. Furthermore, we split the global list of transitions to individual lists for each location of outgoing transitions.

Flow or reset functions are provided in SpaceEx as a list of equations separated by ampersands, as demonstrated in the subsequent example taken from the *platoon_hybrid* model:

```
<flow>
  x1' == x2 &
  x2' == -x3 + u &
  x3' == 1.605*x1 + 4.868*x2 -3.5754*x3 -0.8198*x4 + 0.427*x5 -
         0.045*x6 - 0.1942*x7 + 0.3626*x8 - 0.0946*x9 &
  x4' == x5 &
  x5' == x3 - x6 &
  x6' == 0.8718*x1 + 3.814*x2 -0.0754*x3 + 1.1936*x4 + 3.6258*x5 -
         3.2396*x6 - 0.595*x7+ 0.1294*x8 -0.0796*x9 &
  x7' == x8 &
  x8' == x6 - x9 &
  x9' == 0.7132*x1 + 3.573*x2 - 0.0964*x3 + 0.8472*x4 + 3.2568*x5 -
         0.0876*x6 + 1.2726*x7 + 3.072*x8 - 3.1356*x9 &
  t' == 1
</flow>
```

We separate the equations and represent each one as a tuple of the left-hand side variable name and the right-hand side expression. Variable names are stored as MATLAB strings, while the right-hand-side expressions are stored as *symbolic* expressions of the *Symbolic Math Toolbox*. The Symbolic Math Toolbox also provides powerful manipulation tools such as variable substitution (command `subs`), which are heavily used during the conversion process. The result of the above example is the following struct (symbolic expressions are indicated by curly brackets):

Flow

varNames: [ *"x1" "x2" "x3" "x4" "x5" "x6" "x7" "x8" "x9" "t"* ]
expressions: [ $\{x2\}$ $\{-x3+u\}$ ... $\{1\}$ ]

Invariant and guard sets are similarly defined by a list of equations or inequalities:

```
<invariant>
   t <= 20 &
   min <= u <= max
</invariant>
```

For invariants and guard sets, we convert both sides of each equation or inequality to symbolic expressions. The left side is subtracted by the right side of the equations/inequalities to receive expressions of the form $expr \leq 0$ or $expr = 0$. The result of the above example is

Invariant

inequalities: [ $\{t-20\}$ $\{min-u\}$ $\{u-max\}$ ]
equalities: [ ]

As a result, base components are reformatted into the format shown in Fig. 56.

id
**listOfVar(i)**
**States(i)**
  name
  **Flow**
  **Invariant**
  **Trans(i)**
    destination
    **guard**
    **reset**

Figure 56: Parsed base component template (indexed fields indicate struct arrays).

### Network components

For network components we need to parse the references to other components and perform a variable mapping for each referenced component. Analogously to differential equations in base components, variable mappings in network components are stored using strings and symbolic expressions. We also parse the variables of all components and store their attributes. Please note that *label*-variables are currently ignored, since synchronization label logic is not yet implemented in CORA.

As a result, network components are reformatted into the format shown in Fig. 57.

While loading models with variables named `i, j, I or J`, we discovered that our string to symbolic parser (`str2sym`) automatically replaces them by the constant $\sqrt{-1}$ since MATLAB interprets those as the imaginary unit. As a workaround, we pre-parse all our equations and variable definitions to rename those variables. All names fulfilling the regular expression `i+|j+|I+|J+`

- id
- **listOfVar(i)**
- **Binds(i)**
  - id
  - keys
  - values
  - values_text

Figure 57: Parsed network component template (indexed fields indicate struct arrays).

are lengthened by a letter. The Symbolic Math Toolbox can also substitute other common constants such as `pi`, but does not do so while parsing. It is still recommended to avoid them as variable names.

### D.3.3 Building Component Instances

In the next step, we build the component tree, which represents the hierarchy of all network and base components. An example that demonstrates this process is shown in Fig. 58. The result from the previous conversion step is a list of network and base component templates, where the connections between the list elements are represented as references (binds) between these component templates. To build the component tree, we start from the root component and resolve all of the references to other components. This process is repeated recursively until all leafs of the tree consist of base components, which per definition do not contain any references to other components.

Each time we resolve a reference, we create a base or network component instance from the corresponding template. Note that it is possible that templates are referenced multiple times. In order to create an instance, we have to replace the variable names in the template with the variable names that the parent component specifies for this reference. If the template represents a base component, we rename the variables in the flow function as well as in the equations for the invariant set, the guard sets and the reset functions. Otherwise, if the template represents a network component, we rename the corresponding variables in the outgoing references of the component. Once the component tree is completely build, all instances in the tree use only variables that are defined in the root component, which is crucial for the operations performed in that step.

### D.3.4 Merging Component Instances

In the component tree that was created in the conversion step, each base component instance defines the system dynamics for a subset of the system states. The state vector for the overall system therefore represents a concatenation of the states from the different base component instances. For the component tree that is shown in Fig. 58, the state vector could for example look as follows:

$$\vec{x} = (\underbrace{x_1,\ x_2}_{BC_{1(1)}},\ \underbrace{x_3,\ x_4}_{BC_{1(2)}},\ \underbrace{x_5,\ x_6}_{BC_{1(3)}},\ \underbrace{x_7,\ x_8,\ x_9}_{BC_{2(1)}})^T \tag{56}$$

The component tree therefore represents the overall system as a Compositional Hybrid Automaton. At this point, there exist two different options for the further conversion: Since the 2018 release, CORA provides the class `parallelHybridAutomaton` for the efficient storage and

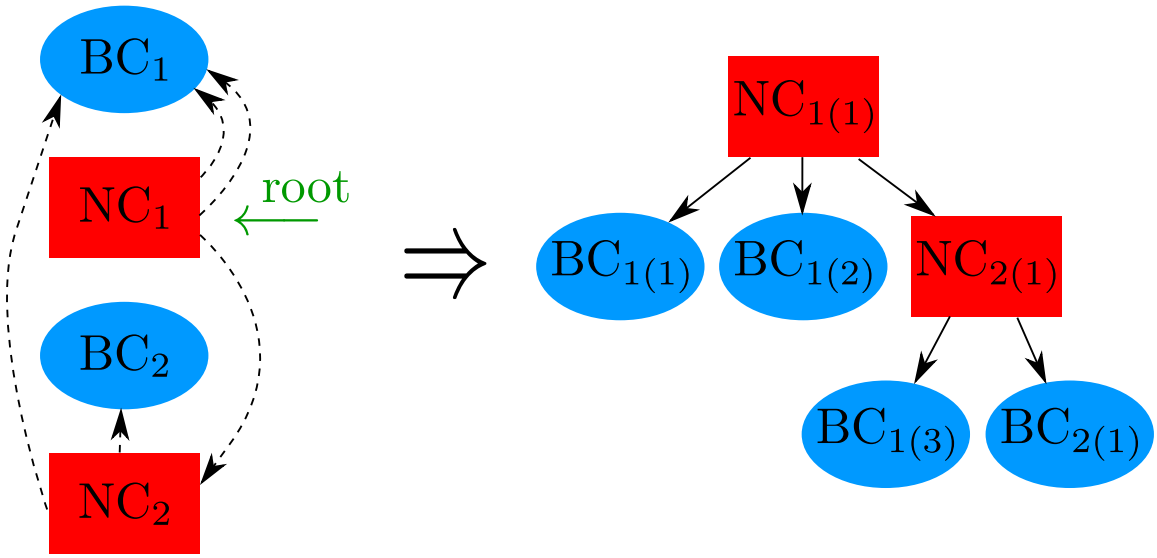



Figure 58: Example for the composition of the component tree. The red nodes represent Network components (NC) and the blue nodes base components (BC). Dashed arrows depict references, while solid arrows represent instantiations.

analysis of Compositional Hybrid Automata (see Sec. 4.3.2). So the SpaceEx model can either be converted to a `parallelHybridAutomaton` object, or to a flat hybrid automaton represented as a `hybridAutomaton` object. In the second case, we have to perform the automaton product, which is shortly described in the remainder of this section.

We have implemented the parallel composition for two base components, which can be applied iteratively to compose a flat hybrid automaton from all components. The product of two instances with discrete state sets $S_1$ and $S_2$ has the state set $S_1 \times S_2$. Thus, we have to compute a new representation for the combined states $\{(s1, s2) | s1 \in S1, s2 \in S2\}$ by combining flow functions, invariants, and transitions. A detailed description of the automaton product and the required operations is provided in [128, Chapter 5] as well as in [104, Def. 2.9].

### D.3.5 Conversion to State-Space Form

Once the composed automaton has been created, we have to convert the descriptions of flow functions, invariant sets, guard sets, and reset functions to a format that can be directly used to create the corresponding CORA objects in the second phase of the conversion process. Subsequently, we describe the required operations for the different parts.

#### Flow Functions

Depending of the type of the flow function, we create different CORA objects. Currently, the converter supports the creation of `linearSys` objects for linear flow functions and `nonlinearSys` objects for nonlinear flow functions. We plan to also include linear as well as nonlinear systems with constant parameters in the future. Up to now, we stored the flow functions as general nonlinear symbolic equations of the form $\dot{x} = f(x, u)$ in the corresponding base components. If the flow function is linear, we have to represent it in the form $\dot{x} = Ax + Bu + c$ in order to be able to construct the `linearSys` object later on. The coefficients for the matrices $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times m}$ can be obtained from the symbolic expressions by computing their partial derivatives:

$$a_{ij} = \frac{\partial f_i(x, u)}{\partial x_j}$$

$$b_{ij} = \frac{\partial f_i(x,u)}{\partial u_j}$$

We compute the partial derivatives with the `jacobian` command from MATLAB's Symbolic Math Toolbox. The constant part $c \in \mathbb{R}^n$ can be easily obtained by substituting all variables with 0:

$$c_i = f_i(0,0)$$

These computations can also be used to check the linearity of a flow function: If the function is linear, then all partial derivatives have to be constant. If a flow fails the linearity test, we create a `nonlinearSys` object instead of a `linearSys` object. This requires the flow equation to be stored in a MATLAB function, which we can easily create by converting symbolic expressions to strings.

#### Reset Functions

Reset functions can be linear or nonlinear and may also depend on inputs, see (49)-(50). CORA represents reset functions with the classes `linearReset` and `nonlinearReset`.

#### Guard Sets and Invariant Sets

The SpaceEx modeling language uses polyhedra for continuous sets. CORA can store polyhedra with the class `polytope`.

Polyhedra can be specified by the coefficients $C \in \mathbb{R}^{p \times n}$, $d \in \mathbb{R}^p$, $C_e \in \mathbb{R}^{q \times n}$, and $d_e \in \mathbb{R}^q$ forming the equation system $Cx \leq d \wedge C_e x = d_e$. We previously stored guards and invariants as symbolic expressions $expr \leq 0$ or $expr = 0$. As for flow functions, the coefficients of $Cx \leq d$ and $C_e x = d_e$ are obtained via partial derivatives and insertion of zeros. Nonlinearity causes an error, since only linear sets are supported by CORA.

### D.4 Creating the CORA model (Phase 2)

In the second phase of the conversion, we generate a MATLAB function that creates a `hybridAutomaton` or `parallelHybridAutomaton` MATLAB object from the parsed SpaceEx model. This function has an identical name as that of the SpaceExModel and is created in `/models/SpaceExConverted/`.

In order to interpret the CORA model in state-space form, each model function starts with an interface specification, presenting which entry of a state or input vector corresponds to which variable in the SpaceEx model. Please find below the example of a chaser spacecraft:

```
%% Interface Specification:
%   This section clarifies the meaning of state & input dimensions
%   by showing their mapping to SpaceEx variable names.

% Component 1 (ChaserSpacecraft):
%  state x := [x; y; vx; vy; t]
%  input u := [uDummy]
```

It is worth noting that CORA does not support zero-input automata. For this reason, we have added a dummy input in the example above.

### D.5 Open Problems

The spaceex2cora converter has already been used in previous ARCH friendly competitions. However, its development is far from being finished. We suggest addressing the following issues in the future:

- **Input constraints:** Input constraints are specified in the SpaceEx format as a part of the invariant set. The input constraints for the converted CORA model should therefore be automatically extracted from the SpaceEx model.
- **Uncertain parameters:** Uncertain system parameters are currently converted to uncertain system inputs for the CORA model. In the future we plan to automatically create `linParamSys` or `nonlinParamSys` objects if uncertain system parameters are present.

## E User Input Validation

In this section, we provide some information about the validation of user inputs for algorithms implemented in CORA. We start with introductory remarks (Appendix E.1), followed by instructions for those who want to use our input validation for the own algorithms (Appendix E.2) and information for developers about the input validation process (Appendix E.3) useful for future feature extensions.

### E.1 Introduction

Algorithms implemented in CORA, see Sec. 4.1, generally take the following three input arguments: A dynamical system `sys`, model parameters `params`, and algorithm parameters `options`. These individual parts have to fit one another: For example, the dimension of the initial set (`params.R0`) has to match the state dimension of a dynamical system (`sys.dim`). Also, each algorithm requires a different set of `options` to execute. To check whether an algorithm is called with suitable arguments for `params` and `options`, we have implemented custom input argument validation. The goal is to have a single call

```
[params,options] = validateOptions(sys,params,options);
```

that validates `params` and `options` for a given system `sys`. If this is not the case, one obtains a clear error message stating what to fix (instead of a cryptic runtime error message). For internal reasons, `params` and `options` are unified into a single `options` struct.

The input argument validation process consists of two main parts:

1. The configuration file for a given system class and algorithm, defining which `params` and `options` can be set and which values they may assume.
2. The validation itself executed in `validateOptions`.

Users who want to use the functionality of `validateOptions` only need to write a configuration file and declare potential default values as well as the set of admissible values for each field of their `params` and `options`, as described in Appendix E.2. Developers who wish to extend to functionality of `validateOptions` will find useful information in Appendix E.3.

Please note that using `validateOptions` is optional and only recommended for algorithms that will be part of a future CORA release! Calling a built-in CORA algorithm does not require any knowledge about `validateOptions`, as one only must follow the interface described in Sec. 4.1. Also, one can disable validation by setting `options.VALIDATE = false` (not recommended).

## E.2 For users

We use configuration files to define all fields of `params` and `options` expected by a given algorithm of a given system class. In this subsection, we will show how to write configuration files, how to define default values for a given field of `params` or `options`, and how to constrain the set of admissible values for a field. To this end, we will also use a running example, alongside the general explanation of the used functions.

Let us start with the configuration file, the purpose of which is to enlist all model parameters `params` and algorithm parameters `options` required for a given algorithm. The configuration file is a function with the following signature:

`[paramsList,optionsList] = config_`*classname*`_`*functionname*

where *classname* needs to match the class of the object, i.e., the result of calling `class(sys)`, passed as the first argument to `validateOptions` and *functionname* is the name of the algorithm. The output arguments are `paramsList` (list of supported model parameters) and `optionsList` (list of supported algorithm parameters).

(Running example: Configuration file – function signature)
We want to write a configuration file for a reachability algorithm for linear systems. Since `class(sys) = linearSys` and our algorithm is implemented in a function called `reach`, our configuration file becomes

```
[paramsList,optionsList] = config_linearSys_reach
```

By convention, we store all configuration files in the directory

`cora/global/functions/helper/dynamics/checkOptions/configfiles`

Let us now look at the content of the configuration file: The first line of the configuration file has to be

```
[paramsList,optionsList] = initDynParameterList();
```

which initializes the output arguments. Next, we add individual fields to `params` and `options` by calling

```
paramsList(end+1,1) = add2list(name,status);
optionsList(end+1,1) = add2list(name,status);
```

The syntax of `add2list` is as follows:

- *name* (char array): name of the field containing at most one dot.
- *status* (char array): either `'mandatory'` (the given field must be defined by the user, otherwise an error is thrown), `'optional'` (the given field can be defined by the user, but it is not required for the computation.) or `'default'` (the given field can be defined by the user, otherwise the default value is taken (see below for how to set default values)).

(Running example: Configuration file – defining model/algorithm parameters)
In our configuration file, we want to add the following mandatory model parameters: An initial set `params.initialSet` and a time horizon `params.timeHorizon`. Additionally, we want to add the algorithm parameter `options.algorithm` (default) and the conditional algorithm parameter `options.optimizeLevel` (mandatory), which is only mandatory for a certain value of `options.algorithm`.
Hence, the content of our configuration file looks as follows:

```
[paramsList,optionsList] = initDynParameterList();
paramsList(end+1,1) = add2list('initialSet','mandatory');
paramsList(end+1,1) = add2list('timeHorizon','mandatory');
optionsList(end+1,1) = add2list('algorithm','default');
optionsList(end+1,1) = add2list('optimizeLevel','mandatory');
```

It goes without saying that identifiers for `params` and `options` should be unique. Important: The behavior for a field like `params.R0` is handled uniformly for *all* algorithms using that specific field. Consequently, please check if there already exists a field with the same name when writing configuration files. Additionally, we must register each field of `params` in the function:

`cora/global/functions/helper/dynamics/checkOptions/isparam.m`

(Running example: Registering fields of model parameters)
We go to the file

`cora/global/functions/helper/dynamics/checkOptions/isparam.m`

and add `'initialSet'` and `'timeHorizon'` to the list of model parameters, so that the output argument `res` becomes `true`.

Next, we want to set the admissible values for the `params` and `options` defined in the configuration file. To this end, we define a set of function handles for each field of `params` and `options`, all of which must evaluate to true for successful validation. All checks are defined in the files

`cora/global/functions/helper/dynamics/checkOptions/checkDynParameterParams.m`
`cora/global/functions/helper/dynamics/checkOptions/checkDynParameterOptions.m`

First, we have a separate `case` block for each field in `params` and `options`, which calls a helper function defining all check functions. For this helper function, we use the convention

```
checks = aux_getChecksParams_name(checks,sys,func,params,options);
checks = aux_getChecksOptions_name(checks,sys,func,params,options);
```

More specifically, we define each check function using the function

`checks(end+1) = add2checks(`*function_handle*`,`*error_id*`);`

where *function_handle* is a function handle returning `true` or `false` and *error_id* is an identifier defined in the file

`cora/global/functions/helper/dynamics/checkOptions/getErrorMessage.m`

Each identifier is mapped to a text explaining why the associated check function failed, i.e., the associated *function_handle* returned `false`.

We recommend to define one-liners and simple expressions directly, but to implement more intricate checks as separate functions in the directory

`cora/global/functions/helper/dynamics/checkOptions/checkFuncs`

and be prefixed by `c_`, the corresponding *error_id* should be an empty char array `''`.

For some fields of `params` or `options`, the set of admissible values is a set of char arrays. We store them in the file

`cora/global/functions/helper/dynamics/checkOptions/getMembers.m`

returning a list of members `memberlist`, against which the user-provided string is checked. In this case, we prefix the *error_id* with `member`.

(Running example: Adding check functions for model parameters)
We want to add the following checks: The initial set `params.initialSet` must be an object of the zonotope class and the time horizon `params.timeHorizon` must be a scalar real value greater than `0`.
First, we open the file

`cora/global/functions/helper/dynamics/checkOptions/checkDynParameterParams.m`

and add `case`-blocks for our fields `initialSet` and `timeHorizon`:

```
case 'initialSet':
    checks = aux_getChecksParams_initialSet(checks,sys,func,params,options);
case 'timeHorizon':
    checks = aux_getChecksParams_timeHorizon(checks,sys,func,params,options);
```

The first helper function is

```
function checks = aux_getChecksParams_initialSet(checks,sys,func,params,options)
    checks(end+1) = add2checks(
      @(val)any(ismember(getMembers('initialSet'),class(val))),'memberinitialSet');
end
```

which defines a function handle that checks whether `initialSet` is an object of the zonotope class. Here, we use the function

`cora/global/functions/helper/dynamics/checkOptions/getMembers.m`

to return the set of admissible values for the class of `initialSet` in the helper function:

```
case 'initialSet':
    memberlist = {'zonotope'};
```

This way, one can easily extend the set of admissible value if another set representation is supported in the future. For our error identifier `'memberinitialSet'` we go to

`cora/global/functions/helper/dynamics/checkOptions/getErrorMessage.m`

and add the identifer to the `switch-case` logic with a suitable error message. Note that many error identifiers for simple checks like `@isscalar` are already defined; re-use them. The helper function for the field `timeHorizon` is

```
function checks = aux_getChecksParams_timeHorizon(checks,sys,func,params,options)
    checks(end+1) = add2checks(@isscalar,'isscalar');
    checks(end+1) = add2checks(@(val)gt(val,0),'gtzero');
end
```

(Running example: Specifying a default value for algorithm parameters)
The algorithm `options.algorithm` must either be `'standard'` or `'optimized'` and the level `options.optimizeLevel` must either be `0`, `1` or `2`. Similary to the last part of the running example, we open the file

`cora/global/functions/helper/dynamics/checkOptions/checkDynParameterOptions.m`

and add `case`-blocks for our fields `algorithm` and `optimizeLevel`:

```
case 'algorithm':
    checks = aux_getChecksOptions_algorithm(checks,sys,func,params,options);
case 'optimizeLevel':
    checks = aux_getChecksOptions_optimizeLevel(checks,sys,func,params,options);
```

The helper functions are implemented as

```
function checks = aux_getChecksOptions_algorithm(checks,sys,func,params,options)
    checks(end+1) = add2checks(
      @(val)any(ismember(getMembers('algorithm'),class(val))),'memberalgorithm');
end
```

and

```
function checks = aux_getChecksOptions_optimizeLevel(checks,sys,func,params,options)
    checks(end+1) = add2checks(@isscalar,'isscalar');
    checks(end+1) = add2checks(@(val)any(val == [0,1,2]),'zero1or2');
end
```

Furthermore, we must alter the file

`cora/global/functions/helper/dynamics/checkOptions/getMembers.m`

by adding an appropriate `case`-block for the admissible values of `algorithm`. Finally, we add a `case`-block for all error identifiers to

`cora/global/functions/helper/dynamics/checkOptions/getErrorMessage.m`

Fields with status *default* are set to their default value specified in the files

`cora/global/functions/helper/dynamics/checkOptions/getDefaultValueParams.m`
`cora/global/functions/helper/dynamics/checkOptions/getDefaultValueOptions.m`

for `params` and `options`, respectively, unless the user specifies a value. The values can either be set directly or implemented using helper functions

`defValue = aux_def_`*name*`(sys,params,options);`

where we can use the information of `sys`, `params`, and `options` to determine a default value.

(Running example: Specifying a default value for algorithm parameters)
The field `algorithm` should have the default value `'standard'`. We open the file

`cora/global/functions/helper/dynamics/checkOptions/getDefaultValueOptions.m`

and add a `case`-block with the desired default value:

```
case 'algorithm'
    defValue = 'standard';
```

Some fields of `params` and `options` are conditionally set, that is, they are only required if another field has a certain value. The conditions are defined in the files

`cora/global/functions/helper/dynamics/checkOptions/getCondfunDynParameterParams.m`
`cora/global/functions/helper/dynamics/checkOptions/getCondfunDynParameterOptions.m`

First, we have a `case`-block for each field in `params` and `options`, which calls a helper function defined by convention as

```
condfun = aux_getCondfunParams_name(sys,func,params,options);
condfun = aux_getCondfunOptions_name(sys,func,params,options);
```

These helper functions return a value `res` that is `true` if a specific condition is met and `false` otherwise.

(Running example: Adding a conditional function for an algorithm parameter)
We want to add the following condition on the field `optimizeLevel`. It should only be required if `options.algorithm` is set to `'optimized'`.
To this end, we open the file

`cora/global/functions/helper/dynamics/checkOptions/getCondfunDynParameterOptions.m`

and add a `case`-block for `optimizeLevel`:

```
case 'optimizeLevel':
    condfun = aux_getCondfunOptions_optimizeLevel;
```

The helper function is implemented as an auxiliary function in the same file:

```
function res = aux_getCondfunOptions_optimizeLevel(sys,func,params,options)
    res = strcmp(options.algorithm,'optimized');
end
```

Finally, let us explicitly list the files users should not have to interact with:

- `add2checks.m`
- `add2list.m`
- `checkDynParameter.m`
- `getCondfunDynParameter.m`
- `getDefaultValue.m`

- `initDynParameterList.m`
- `params2options.m`
- `splitIntoParamsOptions.m`
- `validateOptions.m`

If you encounter any issues regarding these files, please contact the administrator.

### E.3 For developers

In this section, we go through the inner workings of `validateOptions` step by step. First, we check whether `validateOptions` is called internally. This happens because some CORA algorithms call other CORA algorithms. Obviously, once the user input is validated, we do not have to validate it again, as we assume that CORA algorithms are internally parameterized correctly. For internal calls, we only do post-processing and rewrite `params` into `options` for internal use (more on this below). This constitutes a major speed gain, e.g., for hybrid dynamics reachability.

Next, we check whether there exists a configuration file that matches the calling function. If no such file exists, e.g., due to typos or refactoring, we notify the developer/user that `params` and `options` are not checked and we return only the merged `options` struct.

Now, we validate the user inputs for the given `params` and `options`, in this order. We resolve cases where the value for a given field for `params` does not go together with the value for a given field of `options` by blaming the latter. In principle, we read the configuration file, set missing default values, and validate the user-defined values. However, there exist also fields which are conditionally set, i.e., they are only mandatory if some other field has a certain value. As an example, the field `options.krylovError` is only mandatory if the field `options.linAlg` is set to `'krylov'`. This means, we have to first set non-conditional fields, then check the conditions for the remaining fields, and only then check the values of conditional fields.

Both `params` and `options` are checked by the following sequence of steps (the code is extensively documented, please check for more detailed information):

1. Read configuration file. This gives us a blueprint of all potential fields and their statuses.
2. Set missing default values of non-conditional fields.
3. Find redundant fields and fields with a conditional function.
4. Validate non-redundant and non-conditional fields.
5. Check whether conditional fields must be set or not. This can only be done after validating non-redundant and non-conditional fields.
6. For all conditional fields whose condition holds true, set missing default value and validate.
7. Print redundancies encompassing fields which are not defined in the configuration file and fields for which the condition is not met (redundant in this configuration).

After this sequence of steps, we perform a small bit of post-processing in the function `postProcessing`. Here, we rewrite certain fields of `params`, e.g., converting intervals to zonotopes, or determine certain additional fields of `options`, e.g., computing `options.originContained`, which influences the reachable set computation of linear systems. These operations are gathered here to ensure that CORA algorithms know exactly in which format to expect certain variables. Also, some options are fixed to certain values and thus set here (the user should not set them, as only one value is valid).

Finally, we transfer all fields in `params` to `options`. Historically, there was only the `options` struct, so that the code internally only works with one struct. We have introduced the distinction between `params` and `options` to highlight the important difference between model parameters defining the circumstances of the analysis, and algorithm parameters required by the algorithm for execution. It should be a goal of future development to maintain this separation also internally.

# F Licensing

CORA is released under the GPLv3.

# G Disclaimer

The toolbox is primarily for research. We do not guarantee that the code is bug-free.

One needs expert knowledge to obtain optimal results. This tool is prototypical and not all parameters for reachability analysis are automatically set. Not all functions that exist in the software package are explained. Reasons could be that they are experimental or designed for special applications that address a limited audience.

If you have questions or suggestions, please contact us through `https://cora.in.tum.de#contact`.

# H Contributors

All people that have contributed so far are listed in Tab. 35. The table further shows the number of files for each of the different CORA modules that an author contributed to.

Table 35: Number of files that an author contributed to, partitioned by the different modules of CORA.

| | app | contDynamics | contSet | converter | discrDynamics | examples | global | hybridDynamics | matrixSet | models | nn | specification | unitTests |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Matthias Althoff | 0 | 165 | 189 | 7 | 105 | 75 | 51 | 17 | 63 | 56 | 0 | 1 | 164 |
| Mark Wetzlinger | 0 | 83 | 327 | 9 | 0 | 40 | 139 | 62 | 20 | 27 | 0 | 7 | 660 |
| Tobias Ladner | 0 | 20 | 130 | 1 | 0 | 119 | 85 | 2 | 30 | 0 | 78 | 4 | 257 |
| Niklas Kochdumper | 1 | 36 | 277 | 6 | 1 | 58 | 52 | 29 | 3 | 31 | 11 | 58 | 94 |
| Victor Gassmann | 1 | 2 | 97 | 0 | 0 | 4 | 13 | 0 | 3 | 0 | 0 | 0 | 60 |
| Adrian Kulmburg | 0 | 0 | 78 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 57 |
| Dmitry Grebenyuk | 0 | 0 | 66 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 69 |
| Florian Lercher | 0 | 0 | 1 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 41 | 64 |
| Benedikt Seidl | 0 | 0 | 0 | 0 | 0 | 4 | 5 | 0 | 0 | 0 | 0 | 61 | 24 |
| Laura Luetzow | 0 | 20 | 0 | 0 | 0 | 12 | 7 | 0 | 1 | 3 | 0 | 0 | 28 |
| Lukas Koller | 0 | 0 | 0 | 0 | 0 | 3 | 1 | 0 | 0 | 0 | 41 | 0 | 8 |
| Maximilian Perschl | 0 | 0 | 14 | 5 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 18 |
| Viktor Kotsev | 0 | 0 | 21 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 15 |
| Manuel Wendl | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 20 | 0 | 6 |

We also want to thank Aaron Pereira, Farah Atour, Johann Schoepfer, Ahmed El-Guindy, Ivan Brkan, Stefan Liu, Bruno Maione, Carlos Valero, Lukas Schäfer, Zhuoling Li, Mingrui Wang, Sebastian Sigl, Raja Judeh, Severin Prenitzer, Daniel Althoff, Amr Alanwar, Anna Kopetzki, Philipp Gassert, Michael Eichelbeck, Bhaskar Dongare, Martina Hinz, Zeqi Li, Sebastian Mair, Vladimir Popa, Daniel Heß, Leni Rohe, Hanna Krasowski, Wouter Falkena, Michael Kleder, Matt Jacobson, Zhenhai Wang, Victor Charlent, Gerald Würsching, Gerild Pjetri, Tianze Huang, and Hendrik Roehm for individual contributions.

# References


[1] M. Wetzlinger, A. Kulmburg, and M. Althoff, "Inner approximations of reachable sets for nonlinear systems using the minkowski difference," vol. 8, 2024, pp. 2033–2038.

[2] M. Wetzlinger and M. Althoff. (2024) Backward reachability analysis of perturbed continuous-time linear systems using set propagation. ArXiv:2010.11097.

[3] L. Koller, T. Ladner, and M. Althoff, "Set-based training for neural network verification," *arXiv preprint arXiv:2401.14961*, 2024.

[4] M. Wendl, L. Koller, T. Ladner, and M. Althoff, "Training verifiably robust agents using set-based reinforcement learning," *arXiv preprint arXiv:2408.09112*, 2024.

[5] L. Lützow and M. Althoff, "Scalable reachset-conformant identification of linear systems," *IEEE Control Systems Letters*, vol. 8, pp. 520–525, 2024.

[6] ——, "Reachset-conformant system identification," 2024.

[7] F. Lercher and M. Althoff, "Using four-valued signal temporal logic for incremental verification of hybrid systems," in *Computer Aided Verification*, 2024.

[8] M. Wetzlinger, V. Kotsev, A. Kulmburg, and M. Althoff, "Implementation of polyhedral operations in CORA 2024," in *Proc. of the International Workshop on Applied Verification for Continuous and Hybrid Systems*, 2024.

[9] G. Lafferriere, G. J. Pappas, and S. Yovine, "Symbolic reachability computation for families of linear vector fields," *Symbolic Computation*, vol. 32, p. 231–253, 2001.

[10] M. Althoff, "An introduction to CORA 2015," in *Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems*, 2015, p. 120–151.

[11] M. Althoff and D. Grebenyuk, "Implementation of interval arithmetic in CORA 2016," in *Proc. of the 3rd International Workshop on Applied Verification for Continuous and Hybrid Systems*, 2016, p. 91–105.

[12] M. Althoff, D. Grebenyuk, and N. Kochdumper, "Implementation of Taylor models in CORA 2018," in *Proc. of the 5th International Workshop on Applied Verification for Continuous and Hybrid Systems*, 2018, p. 145–173.

[13] J. Lofberg, "YALMIP : a toolbox for modeling and optimization in MATLAB," in *Proc. of the IEEE International Conference on Robotics and Automation*, 2004, p. 284–289.

[14] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable verification of hybrid systems," in *Proc. of the 23rd International Conference on Computer Aided Verification*, ser. LNCS 6806. Springer, 2011, p. 379–395.

[15] M. Althoff, O. Stursberg, and M. Buss, "Reachability analysis of nonlinear systems with uncertain parameters using conservative linearization," in *Proc. of the 47th IEEE Conference on Decision and Control*, 2008, p. 4042–4048.

[16] M. Althoff and G. Frehse, "Combining zonotopes and support functions for efficient reachability analysis of linear systems," in *Proc. of the 55th IEEE Conference on Decision and Control*, 2016, p. 7439–7446.

[17] G. Frehse and M. Althoff, Eds., *ARCH16. 3rd International Workshop on Applied Verification for Continuous and Hybrid Systems*, ser. EPiC Series in Computing, vol. 43, 2017.

[18] ——, *ARCH18. 5th International Workshop on Applied Verification for Continuous and Hybrid Systems*, ser. EasyChair Proceedings in Computing. EasyChair, 2018.

[19] ——, *ARCH19. 6th International Workshop on Applied Verification of Continuous and Hybrid Systems*, ser. EasyChair Proceedings in Computing. EasyChair, 2019.

[20] M. Althoff, S. Bak, D. Cattaruzza, X. Chen, G. Frehse, R. Ray, and S. Schupp, "ARCH-COMP17 category report: Continuous and hybrid systems with linear continuous dynamics," in *Proc. of the 4th International Workshop on Applied Verification for Continuous and Hybrid Systems*, 2017, p. 143–159.

[21] M. Althoff, S. Bak, X. Chen, C. Fan, M. Forets, G. Frehse, N. Kochdumper, Y. Li, S. Mitra, R. Ray, C. Schilling, and S. Schupp, "ARCH-COMP18 category report: Continuous and hybrid systems with linear continuous dynamics," in *Proc. of the 5th International Workshop on Applied Verification for Continuous and Hybrid Systems*, 2018, p. 23–52.

[22] M. Althoff, S. Bak, M. Forets, G. Frehse, N. Kochdumper, R. Ray, C. Schilling, and S. Schupp, "ARCH-COMP19 category report: Continuous and hybrid systems with linear continuous dynamics," in *Proc. of the 6th International Workshop on Applied Verification of Continuous and Hybrid Systems*, ser. EPiC Series in Computing, G. Frehse and M. Althoff, Eds., vol. 61, 2019, p. 14–40.

[23] M. Althoff, S. Bak, Z. Bao, M. Forets, G. Frehse, D. Freire, N. Kochdumper, Y. Li, S. Mitra, R. Ray, C. Schilling, S. Schupp, and M. Wetzlinger, "ARCH-COMP20 category report: Continuous and hybrid systems with linear continuous dynamics," in *Proc. of the 7th International Workshop on Applied Verification of Continuous and Hybrid Systems*, ser. EPiC Series in Computing, G. Frehse and M. Althoff, Eds., vol. 74, 2020, p. 16–48.

[24] X. Chen, M. Althoff, and F. Immler, "ARCH-COMP17 category report: Continuous systems with nonlinear dynamics," in *Proc. of the 4th International Workshop on Applied Verification for Continuous and Hybrid Systems*, 2017, p. 160–169.

[25] F. Immler, M. Althoff, X. Chen, C. Fan, G. Frehse, N. Kochdumper, Y. Li, S. Mitra, M. S. Tomar, and M. Zamani, "ARCH-COMP18 category report: Continuous and hybrid systems with nonlinear dynamics," in *Proc. of the 5th International Workshop on Applied Verification for Continuous and Hybrid Systems*, 2018, p. 53–70.

[26] F. Immler, M. Althoff, L. Benet, A. Chapoutot, X. Chen, M. Forets, L. Geretti, N. Kochdumper, D. P. Sanders, and C. Schilling, "ARCH-COMP19 category report: Continuous and hybrid systems with nonlinear dynamics," in *Proc. of the 6th International Workshop on Applied Verification of Continuous and Hybrid Systems*, ser. EPiC Series in Computing, G. Frehse and M. Althoff, Eds., vol. 61, 2019, p. 41–61.

[27] L. Geretti, J. A. dit Sandretto, M. Althoff, L. Benet, A. Chapoutot, X. Chen, P. Collins, M. Forets, D. Freire, F. Immler, N. Kochdumper, D. P. Sanders, and C. Schilling, "ARCH-COMP20 category report: Continuous and hybrid systems with nonlinear dynamics," in *Proc. of the 7th International Workshop on Applied Verification of Continuous and Hybrid Systems*, ser. EPiC Series in Computing, G. Frehse and M. Althoff, Eds., vol. 74, 2020, p. 49–75.

[28] A. Kulmburg, I. Brkan, and M. Althoff, "Search-based and stochastic solutions to the zonotope and ellipsotope containment problems," in *2024 European Control Conference*, 2024, pp. 1057–1064.

[29] A. Kulmburg and M. Althoff, "On the co-np-completeness of the zonotope containment problem," *European Journal of Control*, vol. 62, pp. 84–91, 2021.

[30] A.-K. Kopetzki, B. Schürmann, and M. Althoff, "Methods for order reduction of zonotopes," in *Proc. of the 56th IEEE Conference on Decision and Control*, 2017, p. 5626–5633.

[31] C. Combastel, "A state bounding observer based on zonotopes," in *Proc. of the European Control Conference*, 2003, p. 2589–2594.

[32] A. Girard, "Reachability of uncertain linear systems using zonotopes," in *Hybrid Systems: Computation and Control*, ser. LNCS 3414. Springer, 2005, p. 291–305.

[33] M. Althoff, "Reachability analysis and its application to the safety assessment of autonomous cars," Dissertation, Technische Universität München, 2010, http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20100715-963752-1-4.

[34] J. K. Scott, D. M. Raimondo, G. R. Marseglia, and R. D. Braatz, "Constrained zonotopes: A new tool for set-based estimation and fault detection," *Automatica*, vol. 69, p. 126–136, 2016.

[35] A. A. Kurzhanskiy and P. Varaiya, "Ellipsoidal toolbox," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-46, 2006. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-46.html

[36] G. M. Ziegler, *Lectures on Polytopes*, ser. Graduate Texts in Mathematics. Springer, 1995.

[37] V. Kaibel and M. E. Pfetsch, *Algebra, Geometry and Software Systems*. Springer, 2003, ch. Some Algorithmic Problems in Polytope Theory, p. 23–47.

[38] M. Althoff, "Reachability analysis of nonlinear systems using conservative polynomialization and non-convex sets," in *Hybrid Systems: Computation and Control*, 2013, p. 173–182.

[39] N. Kochdumper and M. Althoff, "Sparse polynomial zonotopes: A novel set representation for reachability analysis," *IEEE Transactions on Automatic Control*, vol. 66, no. 9, p. 4043–4058, 2021.

[40] N. Kochdumper, B. Schürmann, and M. Althoff, "Utilizing dependencies to obtain subsets of reachable sets," in *Proc. of the 23rd ACM International Conference on Hybrid Systems: Computation and Control*, 2020, article No. 1.

[41] N. Kochdumper, P. Gassert, and M. Althoff, "Verification of collision avoidance for CommonRoad traffic scenarios," in *8th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH21)*, ser. EPiC Series in Computing, G. Frehse and M. Althoff, Eds., vol. 80. EasyChair, 2021, p. 184–194. [Online]. Available: https://easychair.org/publications/paper/cqFP

[42] M. Althoff and B. H. Krogh, "Zonotope bundles for the efficient computation of reachable sets," in *Proc. of the 50th IEEE Conference on Decision and Control*, 2011, p. 6814–6821.

[43] T. Netzer, "Spectrahedra and their shadows," Ph.D. dissertation, Universität Leipzig, 2011.

[44] M. Althoff, O. Stursberg, and M. Buss, "Safety assessment for stochastic linear systems using enclosing hulls of probability density functions," in *Proc. of the European Control Conference*, 2009, p. 625–630.

[45] D. Berleant, "Automatically verified reasoning with both intervals and probability density functions," *Interval Computations*, vol. 2, p. 48–70, 1993.

[46] L. Jaulin, M. Kieffer, and O. Didrit, *Applied Interval Analysis*. Springer, 2006.

[47] K. Makino and M. Berz, "Taylor models and other validated functional inclusion methods," *International Journal of Pure and Applied Mathematics*, vol. 4, no. 4, p. 379–456, 2003.

[48] L. H. de Figueiredo and J. Stolfi, "Affine arithmetic: Concepts and applications," *Numerical Algorithms*, vol. 37, no. 1-4, p. 147–158, 2004. [Online]. Available: http://link.springer.com/10.1023/B:NUMA.0000049462.70970.b6

[49] M. Berz and G. Hoffstätter, "Computation and application of Taylor polynomials with interval remainder bounds," *Reliable Computing*, vol. 4, p. 83–97, 1998.

[50] K. Makino and M. Berz, "Remainder differential algebras and their applications," in *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, 1996, p. 63–74.

[51] ——, "Rigorous integration of flows and ODEs using Taylor models," in *Proc. of Symbolic-Numeric Computation*, 2009, p. 79–84.

[52] W. Kühn, *Mathematical Visualization*. Springer, 1998, ch. Zonotope Dynamics in Numerical Quality Control, p. 125–134.

[53] H. Roehm, J. Oehlerking, M. Woehrle, and M. Althoff, "Reachset conformance testing of hybrid automata," in *Proc. of Hybrid Systems: Computation and Control*, 2016, p. 277–286.

[54] M. Althoff and J. J. Rath, "Comparison of guaranteed state estimators for linear time-invariant systems," *Automatica*, vol. 130, 2021, article no. 109662.

[55] M. Althoff, "Guaranteed state estimation in CORA 2021," in *Proc. of the 8th International Workshop on Applied Verification of Continuous and Hybrid Systems*, ser. EPiC Series in Computing, G. Frehse and M. Althoff, Eds., vol. 80. EasyChair, 2021, p. 161–175. [Online]. Available: https://easychair.org/publications/paper/hMPK

[56] ——, "Checking and establishing reachset conformance in cora 2023," in *Proc. of 10th International Workshop on Applied Verification of Continuous and Hybrid Systems*, ser. EPiC Series in Computing. EasyChair, 2023.

[57] H. Roehm, J. Oehlerking, M. Woehrle, and M. Althoff, "Model conformance for cyber-physical systems: A survey," *ACM Transactions on Cyber-Physical Systems*, vol. 3, no. 3, p. Article 30, 2019.

[58] H. Roehm, A. Rausch, and M. Althoff, "Reachset conformance and automatic model adaptation for hybrid systems," *Mathematics*, vol. 10, no. 19, p. Article 3567, 2022. [Online]. Available: https://www.mdpi.com/2227-7390/10/19/3567

[59] H. Roehm, J. Oehlerking, T. Heinz, and M. Althoff, "STL model checking of continuous and hybrid systems," in *Proc. of the 14th International Symposium on Automated Technology for Verification and Analysis*, 2016, p. 412–427.

[60] M. Althoff and J. M. Dolan, "Reachability computation of low-order models for the safety verification of high-order road vehicle models," in *Proc. of the American Control Conference*, 2012, p. 3559–3566.

[61] S. B. Liu, H. Roehm, C. Heinzemann, I. Lütkebohle, J. Oehlerking, and M. Althoff, "Provably safe motion of mobile robots in human environments," in *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2017, p. 1351–1357.

[62] S. B. Liu and M. Althoff, "Reachset conformance of forward dynamic models for the formal analysis of robots," in *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2018, p. 370–376.

[63] ——, "Online verification of impact-force-limiting control for physical human-robot interaction," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2021, p. 777–783.

[64] N. Kochdumper, A. Tarraf, M. Rechmal, M. Olbrich, L. Hedrich, and M. Althoff, "Establishing reachset conformance for the formal analysis of analog circuits," in *Proc. of the 25th Asia and South Pacific Design Automation Conference*, 2020, p. 199–204.

[65] A. Donzé and G. Frehse, "Modular, hierarchical models of control systems in SpaceEx," in *Proc. of the European Control Conference*, 2013, p. 4244–4251.

[66] A. Girard, C. Le Guernic, and O. Maler, "Efficient computation of reachable sets of linear time-invariant systems with inputs," in *Hybrid Systems: Computation and Control*, ser. LNCS 3927. Springer, 2006, p. 257–271.

[67] S. Bogomolov, M. Forets, G. Frehse, F. Viry, A. Podelski, and C. Schilling, "Reach set approximation through decomposition with low-dimensional sets and high-dimensional matrices," in *Proc. of the 21st International Conference on Hybrid Systems: Computation and Control*, 2018, p. 41–50.

[68] M. Althoff, "Reachability analysis of large linear systems with uncertain inputs in the Krylov subspace," *IEEE Transactions on Automatic Control*, vol. 65, no. 2, p. 477–492, 2020.

[69] M. Wetzlinger, N. Kochdumper, and M. Althoff, "Adaptive parameter tuning for reachability analysis of linear systems," in *Proc. of the 59th IEEE Conference on Decision and Control*, 2020.

[70] M. Althoff, O. Stursberg, and M. Buss, "Reachability analysis of linear systems with uncertain parameters and inputs," in *Proc. of the 46th IEEE Conference on Decision and Control*, 2007, p. 726–732.

[71] B. Schürmann, N. Kochdumper, and M. Althoff, "Reachset model predictive control for disturbed nonlinear systems," in *Proc. of the 57th IEEE Conference on Decision and Control*, 2018, p. 3463–3470.

[72] T. Alamo, J. M. Bravo, and E. F. Camacho, "Guaranteed state estimation by zonotopes," *Automatica*, vol. 41, no. 6, p. 1035–1043, 2005.

[73] J. M. Bravo, T. Alamo, and E. F. Camacho, "Bounded error identification of systems with time-varying parameters," *IEEE Transactions on Automatic Control*, vol. 51, no. 7, p. 1144–1150, 2006.

[74] Y. Wang, V. Puig, and G. Cembrano, "Set-membership approach and Kalman observer based on zonotopes for discrete-time descriptor systems," *Automatica*, vol. 93, p. 435–443, 2018.

[75] V. T. H. Le, C. Stoica, T. Alamo, E. F. Camacho, and D. Dumur, "Zonotopic guaranteed state estimation for uncertain systems," *Automatica*, vol. 49, no. 11, p. 3418–3424, 2013.

[76] ——, "Zonotope-based set-membership estimation for multi-output uncertain systems," in *Proc. of the IEEE International Symposium on Intelligent Control*, 2013, p. 212–217.

[77] Y. Wang, T. Alamo, V. Puig, and G. Cembrano, "A distributed set-membership approach based on zonotopes for interconnected systems," in *Proc. of the IEEE Conference on Decision and Control*, 2018, p. 668–673.

[78] Y. Wang, Z. Wang, V. Puig, and G. Cembrano, "Zonotopic set-membership state estimation for discrete-time descriptor LPV systems," *IEEE Transactions on Automatic Control*, vol. 64, no. 5, p. 2092–2099, 2019.

[79] A. Alanwar, V. Gassmann, X. He, H. Said, H. Sandberg, K. H. Johansson, and M. Althoff. (2020) Privacy preserving set-based estimation using partially homomorphic encryption. ArXiv:2010.11097.

[80] S. Gollamudi, S. Nagaraj, S. Kapoor, and Y. F. Huang, "Set-membership state estimation with optimal bounding ellipsoids," in *Proc. of the International Symposium on Information Theory and its Applications*, 1996, p. 262–265.

[81] Y. Liu, Y. Zhao, and F. Wu, "Ellipsoidal state-bounding-based set-membership estimation for linear system with unknown-but-bounded disturbances," *IET Control Theory & Applications*, vol. 10, no. 4, p. 431–442, 2016.

[82] C. Combastel, "Zonotopes and Kalman observers: Gain optimality under distinct uncertainty paradigms and robust convergence," *Automatica*, vol. 55, p. 265–273, 2015.

[83] Y. Wang, M. Zhou, V. Puig, G. Cembrano, and Z. Wang, "Zonotopic fault detection observer with $\mathcal{H}_-$ performance," in *Proc. of the 36th IEEE Chinese Control Conference*, 2017, p. 7230–7235.

[84] N. Loukkas, J. J. Martinez, and N. Meslem, "Set-membership observer design based on ellipsoidal invariant sets," *IFAC-PapersOnLine*, vol. 50, no. 1, p. 6471–6476, 2017.

[85] J. J. Martinez, N. Loukkas, and N. Meslem, "H-infinity set-membership observer design for discrete-time LPV systems," *International Journal of Control*, vol. 93, no. 10, p. 2314–2325, 2020.

[86] W. Tang, Z. Wang, Y. Wang, T. Raïssi, and Y. Shen, "Interval estimation methods for discrete-time linear time-invariant systems," *IEEE Transactions on Automatic Control*, vol. 64, no. 11, p. 4717–4724, 2019.

[87] S. B. Liu, B. Schürmann, and M. Althoff, "Guarantees for real robotic systems: Unifying formal controller synthesis and reachset-conformant identification," *IEEE Transactions on Robotics*, 2023, early access.

[88] C. W. Gardiner, *Handbook of Stochastic Methods: For Physics, Chemistry and the Natural Sciences*, H. Haken, Ed. Springer, 1983.

[89] L. Lützow and M. Althoff, "Reachability analysis of ARMAX models," in *Proc. of the 62nd IEEE Conference on Decision and Control*, 2023.

[90] M. Wetzlinger, A. Kulmburg, and M. Althoff, "Adaptive parameter tuning for reachability analysis of nonlinear systems," in *Proc. of the 24th International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '21. Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3447928.3456643

[91] N. Kochdumper and M. Althoff, "Computing non-convex inner-approximations of reachable sets for nonlinear continuous systems," in *Proc. of the 59th IEEE Conference on Decision and Control*, 2020, p. 2130–2137.

[92] E. Goubault and S. Putot, "Robust under-approximations and application to reachability of non-linear control systems with disturbances," *IEEE Control Systems Letters*, vol. 4, no. 4, pp. 928–933, 2020.

[93] ——, "Forward inner-approximated reachability of non-linear continuous systems," in *Proc. of the 20th International Conference on Hybrid Systems: Computation and Control*, 2017, p. 1–10.

[94] U. M. Ascher and L. R. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM: Society for Industrial and Applied Mathematics, 1998.

[95] K. E. Brenan, S. L. Campbell, and L. R. Petzold, *Numerical Solution of Initial Value Problems in Differential-Algebraic Equations*. North-Holland, 1989.

[96] M. Althoff and B. H. Krogh, "Reachability analysis of nonlinear differential-algebraic systems," *IEEE Transactions on Automatic Control*, vol. 59, no. 2, p. 371–383, 2014.

[97] N. Kochdumper, C. Schilling, M. Althoff, and S. Bak, "Open- and closed-loop neural network verification using polynomial zonotopes," in *Proc. of NASA Formal Methods*, 2023.

[98] T. Ladner and M. Althoff, "Automatic abstraction refinement in neural network verification using sensitivity analysis," in *Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control*, 2023, pp. 1–13.

[99] M. Althoff, O. Stursberg, and M. Buss, "Computing reachable sets of hybrid systems using a combination of zonotopes and polytopes," *Nonlinear Analysis: Hybrid Systems*, vol. 4, no. 2, p. 233–249, 2010.

[100] A. Girard and C. Le Guernic, "Zonotope/hyperplane intersection for hybrid systems reachability analysis," in *Proc. of Hybrid Systems: Computation and Control*, ser. LNCS 4981. Springer, 2008, p. 215–228.

[101] M. Althoff and B. H. Krogh, "Avoiding geometric intersection operations in reachability analysis of hybrid systems," in *Hybrid Systems: Computation and Control*, 2012, p. 45–54.

[102] S. Bak, S. Bogomolov, and M. Althoff, "Time-triggered conversion of guards for reachability analysis of hybrid automata," in *Proc. of the 15th International Conference on Formal Modelling and Analysis of Timed Systems*, 2017, p. 133–150.

[103] N. Kochdumper and M. Althoff, "Reachability analysis for hybrid systems with nonlinear guard sets," in *Proc. of the 23rd ACM International Conference on Hybrid Systems: Computation and Control*, 2020, article No. 2.

[104] G. Frehse, "Compositional verification of hybrid systems using simulation relations," Ph.D. dissertation, Radboud Universiteit Nijmegen, 2005.

[105] H.-S. L. Lee, M. Althoff, S. Hoelldampf, M. Olbrich, and E. Barke, "Automated generation of hybrid system models for reachability analysis of nonlinear analog circuits," in *Proc. of the 20th Asia and South Pacific Design Automation Conference*, 2015, p. 725–730.

[106] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Springer, 2009.

[107] M. Althoff, O. Stursberg, and M. Buss, "Model-based probabilistic collision detection in autonomous driving," *IEEE Transactions on Intelligent Transportation Systems*, vol. 10, no. 2, p. 299–310, 2009.

[108] M. Althoff and A. Mergel, "Comparison of Markov chain abstraction and Monte Carlo simulation for the safety assessment of autonomous cars," *IEEE Transactions on Intelligent Transportation Systems*, vol. 12, no. 4, p. 1237–1247, 2011.

[109] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *Computer Aided Verification*. Springer, 2017, p. 97–117.

[110] F. Blanchini, "Set invariance in control," *Automatica*, vol. 35, no. 11, p. 1747–1767, 1999.

[111] K. Makino and M. Berz, "Verified global optimization with Taylor model based range bounders," *Transactions on Computers*, vol. 4, no. 11, p. 1611–1618, 2005.

[112] E. Kaucher, *Interval Analysis in the Extended Interval Space IR*, 1980, p. 33–49.

[113] G. Trombettoni, Y. Papegay, G. Chabert, and O. Pourtallier, "A box-consistency contractor based on extremal functions," in *Principles and Practice of Constraint Programming*, D. Cohen, Ed., 2010, p. 491–498.

[114] S. Minopoli and G. Frehse, "SL2SX translator: From simulink to spaceex models," in *Proc. of the 19th International Conference on Hybrid Systems: Computation and Control*, 2016, p. 93–98.

[115] N. Kekatos, M. Forets, and G. Frehse, "Constructing verification models of nonlinear simulink systems via syntactic hybridization," in *Proc. of the 56th IEEE Conference on Decision and Control*, 2017, p. 1788–1795.

[116] S. Bak, S. Bogomolov, and T. T. Johnson, "HYST: a source transformation and translation tool for hybrid automaton models," in *Proc. of the 18th International Conference on Hybrid Systems: Computation and Control*, 2015.

[117] S. Cotton, G. Frehse, and O. Lebeltel. (2010) The spaceex modeling language. [Online]. Available: http://spaceex.imag.fr/sites/default/files/spaceex_modeling_language_0.pdf

[118] M. Althoff, B. H. Krogh, and O. Stursberg, *Modeling, Design, and Simulation of Systems with Uncertainties*. Springer, 2011, ch. Analyzing Reachability of Linear Dynamic Systems with Parametric Uncertainties, p. 69–94.

[119] M. Althoff and J. M. Dolan, "Set-based computation of vehicle behaviors for the online verification of autonomous vehicles," in *Proc. of the 14th IEEE Conference on Intelligent Transportation Systems*, 2011, p. 1162–1167.

[120] ——, "Online verification of automated road vehicles using reachability analysis," *IEEE Transactions on Robotics*, vol. 30, no. 4, p. 903–918, 2014.

[121] J. M. Bravo, T. Alamo, and E. F. Camacho, "Robust MPC of constrained discrete-time nonlinear systems based on approximated reachable sets," *Automatica*, vol. 42, p. 1745–1751, 2006.

[122] M. Althoff, M. Cvetković, and M. Ilić, "Transient stability analysis by reachable set computation," in *Proc. of the IEEE PES Conference on Innovative Smart Grid Technologies Europe*, 2012, p. 1–8.

[123] M. Althoff, "Formal and compositional analysis of power systems using reachable sets," *IEEE Transactions on Power Systems*, vol. 29, no. 5, p. 2270–2280, 2014.

[124] H. A. van der Schaft AND Schumacher, *An Introduction to Hybrid Dynamical Systems*. Springer, 2000.

[125] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[126] V. Gaßmann and M. Althoff, "Scalable zonotope-ellipsoid conversions using the Euclidean zonotope norm," in *Proc. of the American Control Conference*, 2020, p. 4715–4721.

[127] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, p. 231–274, 1987.

[128] E. A. Lee and S. A. Seshia, *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press, 2016.