CORA 2018 Manual

Matthias Althoff and Niklas Kochdumper

Technische Universität München, 85748 Garching, Germany

Abstract

The philosophy, architecture, and capabilities of the COntinuous Reachability Analyzer (CORA) are presented. CORA is a toolbox that integrates various vector and matrix set representations and operations on these set representations as well as reachability algorithms of various dynamic system classes. The software is designed such that set representations can be exchanged without having to modify the code for reachability analysis. CORA has a modular design, making it possible to use the capabilities of the various set representations for other purposes besides reachability analysis. The toolbox is designed using the object oriented paradigm, such that users can safely use methods without concerning themselves with detailed information hidden inside the object. Since the toolbox is written in MATLAB, the installation and use is platform independent. CORA is released under the GPLv3.

Contents

1 What's new compared to CORA 2016?

It is our pleasure to present many new features for CORA 2018. The subsequent list is non-exhaustive and unsorted:

- **Reading SpaceEx format**: One can now read SpaceEx models (see Sec. 14), which have become the quasi-standard for formal verification tools of hybrid systems. This also has the advantage that one can use the SpaceEx model editor[1] for modeling hybrid systems.
- **Parallel hybrid automata**: It is infeasible to model larger hybrid systems using a single hybrid automaton. It is now possible to specify parallel hybrid automata so that it is no longer required to model a system by a single hybrid automaton. For analysis purposes, we assemble the dynamics of parallel hybrid automata on-the-fly as described e.g., in [1].
- **New zonotope reduction methods**: New methods for the order reduction of zonotopes presented in [2] are now available in CORA.
- **Lazy symbolic computations**: CORA performs symbolic computations for nonlinear systems, e.g., to linearize them for various linearization points. These computations used to be performed for each reachability analysis. Now these time-consuming computations are only performed if the model files are changed or options concerning the symbolic computations are modified.
- **Taylor models**: In our new version, we have realized a class to compute with Taylor models as described in [3].
- **Affine arithmetic**: As a byproduct of the Taylor model implementation, we have also implemented an affine arithmetic when the number of noise terms does not exceed the system dimension, see [3, Sec. 3].
- **Constrained zonotope**: This new set representation from [4] is as general as polytopes, but makes it possible to use lazy computations.
- **Discrete time models**: It is now possible to compute reachable sets of discrete time models; see Sec. 9.5. We have not implemented a class for linear discrete time models, since this is too trivial.
- **More compact implementation**: We have integrated the class `linVarSys` into the class `linParamSys`. Also, we have unified all symbolic computations, which are now inherited from all other classes for continuous dynamics.
- **Miscellaneous**: There are many other interesting improvements: Better organization of models (now under /models), directly changing parameters for nonlinear systems during reachability analysis, better controlling simplifications of symbolic expressions, more unit tests, vector and matrix norms for vector and matrix sets, extending linear systems to affine systems $\dot{x} = Ax + Bu + c$, etc.

2 Philosophy and Architecture

The **CO**ntinuous **R**eachability **A**nalyzer (CORA)[2] 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,

[1] spaceex.imag.fr/download-6

[2] `https://www6.in.tum.de/Main/SoftwareCORA`

parameter-varying systems, etc.) and hybrid dynamics combining the aforementioned continuous dynamics with discrete dynamics. 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 $t = r$ 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

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

CORA solely supports over-approximative computation of reachable sets since (a) exact reachable sets cannot be computed for most system classes [5] and (b) over-approximative computations qualify for formal 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 enables one to construct one's own reachable set computation in a relatively short amount of time. This is achieved by the following design choices:

- CORA is built for 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 one is interested in without redesigning the whole code. Operator overloading makes it possible to write formulas that look almost identical to the ones derived in scientific papers and thus reduce programming errors. Most of the information for each class is hidden and is not relevant to users of the toolbox. Most classes use identical methods so that set representations and dynamic systems can be effortlessly replaced.
- CORA interfaces with the established toolbox MPT[3], which is also written in MATLAB. Results of CORA can be easily transferred to this toolbox and vice versa. We are currently supporting version 2 and 3 of the MPT.

Of course, it is also possible to use CORA as it is to conduct 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. 12). 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 is investigated as part of 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 [3, 6, 7].

3 Installation

The software does not require any installation, except that the path for CORA has to be set in MATLAB. Besides CORA, the MPT toolbox has to be downloaded and included in the MATLAB path: `http://people.ee.ethz.ch/~mpt/3/`. If the new installation routine of the

[3] `http://control.ee.ethz.ch/~mpt/2/`

MPT is used, it is no longer required to manually include MPT in the MATLAB path. MPT is designed for parametric optimization, computational geometry and model predictive control. CORA only uses the computational geometry capabilities for polytopes.

CORA also requires the **symbolic math toolbox** in MATLAB.

To check whether you correctly included all files in the MATLAB path, type `runTestSuite` in the MATLAB workspace to run all unit tests (see Sec. 13).

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 [8] have been integrated to a certain extent.

Importing SpaceEx Models CORA can now read SpaceEx models as described in Sec. 14. This has two major benefits: First, SpaceEx has become the quasi-standard for model exchange between tools for the formal verification of hybrid systems (see ARCH friendly competition in Sec. 5) so that many model files in this format are available. Second, there exists a graphical model editor for Space Ex models briefly presented in Sec. 14.1 helping non-experts to model hybrid systems more easily.

CORA/SX CORA code for computing reachable sets of nonlinear systems is now 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 [9] has been implemented. Also, the zonotope class from CORA is now available in CORA/SX, making efficient computations for switched linear systems possible as described in [10].

5 CORA@ARCH

CORA has participated in the ARCH[4] friendly competitions since the first competition in 2017. Results of the competition can be found in the yearly ARCH proceedings [11, 12]. In particular, CORA has participated in the linear systems category [13,14] and the nonlinear systems category [15, 16]; CORA/SX has participated in the same categories in 2018 [14, 16].

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/.

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

6 Architecture

The architecture of CORA can essentially be grouped into the parts presented in Fig. 1 using UML[5]: Classes for set representations (Sec. 7), classes for matrix set representations (Sec.

[4] Applied Verification for Continuous and Hybrid Systems
[5] `http://www.uml.org/`

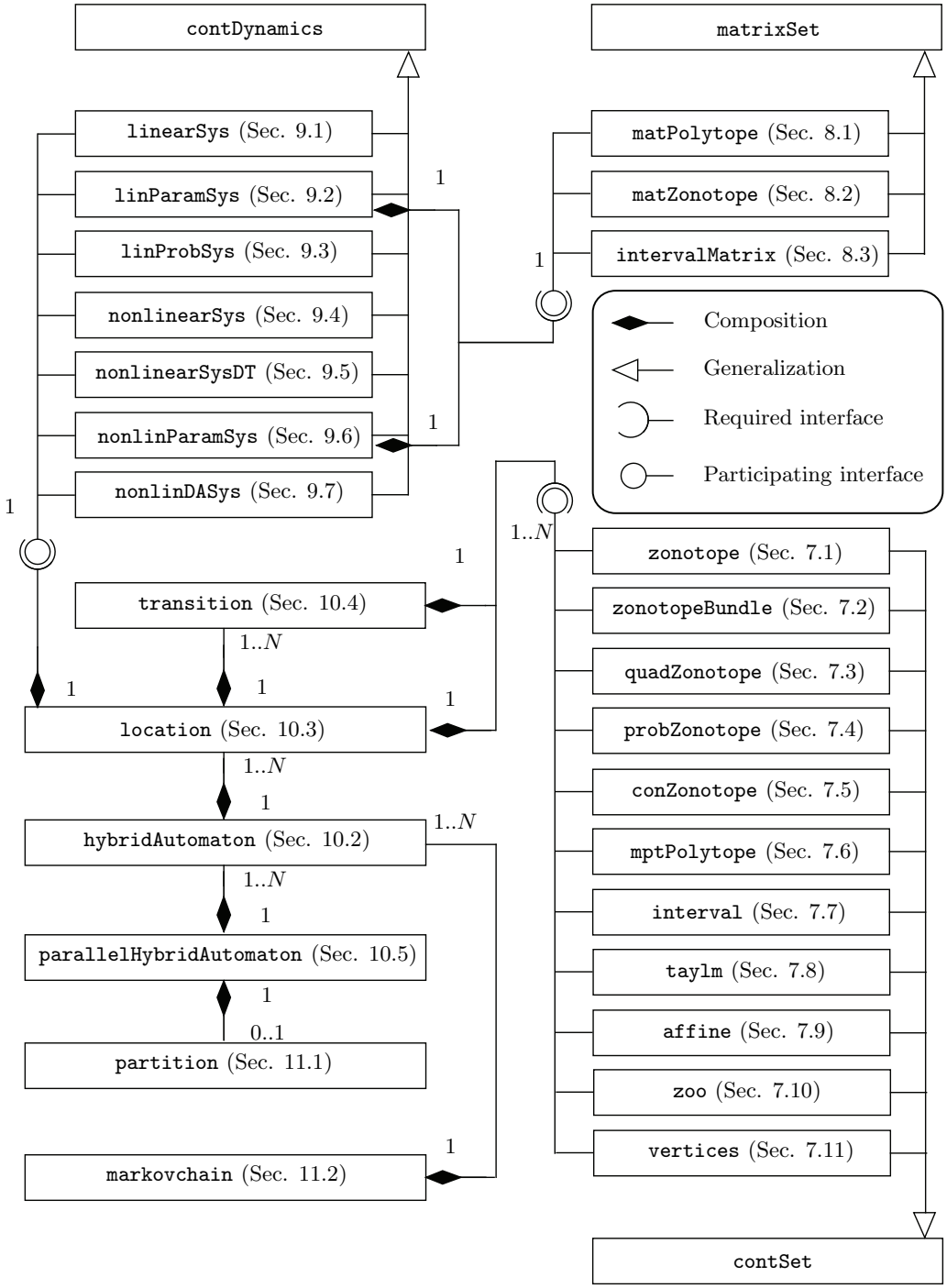


Figure 1: Unified Modeling Language (UML) class diagram of CORA.

8), classes for the analysis of continuous dynamics (Sec. 9), classes for the analysis of hybrid dynamics (Sec. 10), and classes for the abstraction to discrete systems (Sec. 11).

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 `location` object has a 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. 10.

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`). The reason why less general systems are not removed is because very efficient algorithms exist for those systems that are not applicable to more general systems.

7 Set Representations and Operations

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 makes it possible to run efficient and scalable operations on them. CORA provides a palette of set representations that are explained in detail throughout this section. Table 1 shows the supported conversions between the set representations. In order to convert a set, it is sufficient to pass the current set object to the class constructor of the target set representation, as demonstrated by the following code example:

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

% convert to other set representations
int = interval(zono);           % over-approximation
poly = mptPolytope(zono);       % exact conversion
```

Table 1: 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.

	zono	**zB**	**qZ**	**pZ**	**cZ**	**poly**	**int**	**tay**	**vert**
zonotope (zono, Sec. 7.1)	-	e	e		e	e	o		e
zonotopeBundle (zB, Sec. 7.2)		-			e	e	o		e
quadZonotope (qZ, Sec. 7.3)	o		-				o		
probZonotope (pZ, Sec. 7.4)	o			-					
conZonotope (cZ, Sec. 7.5)	o				-	e	o		e
mptPolytope (poly, Sec. 7.6)					e	-	o		e
interval (int, Sec. 7.7)	e				e	e	-		e
taylm (tay, Sec. 7.8)							o	-	
vertices (vert, Sec. 7.11)	o						o		-

Important operations for sets are:

- `display`: Displays the parameters of the set in the MATLAB workspace.
- `plot`: Plots a two-dimensional projection of a set in the current MATLAB figure.
- `plotFilled`: Like `plot`, but fills the plot with a specified color.
- `mtimes`: Overloads the '`*`' operator for the multiplication of various objects with a set. For instance if M is a matrix of proper dimension and Z is a zonotope, `M * Z` returns the linear map $\{Mx | x \in Z\}$.
- `plus`: Overloads the '`+`' operator for the addition of various objects with a set. For instance if $Z1$ and $Z2$ are zonotopes of proper dimension, `Z1 + Z2` returns the Minkowski

sum $\{x + y | x \in Z1, y \in Z2\}$.

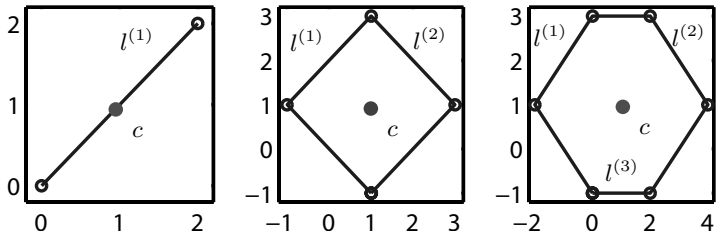
- `interval`: Returns an interval that encloses the set (see Sec. 7.7).

7.1 Zonotopes

A zonotope is a geometric object in $\mathbb{R}^n$. Zonotopes are parameterized by a center $c \in \mathbb{R}^n$ and generators $g^{(i)} \in \mathbb{R}^n$ and defined as

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

We write in short $\mathcal{Z} = (c, g^{(1)}, \ldots, g^{(p)})$. A zonotope can be interpreted as the Minkowski addition of line segments $l^{(i)} = [-1, 1]g^{(i)}$ and is visualized step-by-step in a two-dimensional vector space in Fig. 2. Zonotopes are a compact way of representing sets in high dimensions. More importantly, operations required for reachability analysis, such as linear maps $M \otimes \mathcal{Z} := \{Mz | z \in \mathcal{Z}\}$ ($M \in \mathbb{R}^{q \times n}$) and Minkowski addition $\mathcal{Z}_1 \oplus \mathcal{Z}_2 := \{z_1 + z_2 | z_1 \in \mathcal{Z}_1, z_2 \in \mathcal{Z}_2\}$ can be computed efficiently and exactly, and others such as convex hull computation can be tightly over-approximated [17].



(a) $c \oplus l^{(1)}$ (b) $c \oplus l^{(1)} \oplus l^{(2)}$ (c) $c \oplus \ldots \oplus l^{(3)}$

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

We support the following methods for zonotopes:

- `box` – computes an enclosing axis-aligned box in generator representation.
- `cartesianProduct` – returns the Cartesian product of two zonotopes.
- `center` – returns the center of the zonotope.
- `constrSat` – checks if all values of a zonotope satisfy the constraint $Cx <= d$, $C \in \mathbb{R}^{m \times n}$, $d \in \mathbb{R}^m$.
- `containsPoint` – determines if a point is inside a zonotope.
- `conZonotope` – converts a zonotope object into a constrained zonotope object.
- `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.
- `dim` – returns the dimension of a zonotope in the sense that the rank of the generator matrix is computed.
- `display` – standard method, see Sec. 7.
- `enclose` – generates a zonotope that encloses two zonotopes of equal dimension according to [18, Equation 2.2 + subsequent extension].

- `enclosingPolytope` – converts a zonotope to a polytope representation in an over-approximative way to save computational time. The technique can be influenced by options, but most techniques are inspired by [18, Sec. 2.5.6].
- `enlarge` – enlarges the generators of a zonotope by a vector of factors for each dimension.
- `generators` – returns the generators of a zonotope as a matrix whose column vectors are the generators.
- `in` – determines if a zonotope is enclosed by another zonotope.
- `inParallelotope` – checks if a zonotope is a subset of a parallelotope, where the latter is represented as a zonotope.
- `interval` – standard method, see Sec. 7. More details can be found in [18, Proposition 2.2].
- `isempty` – returns 1 if a zonotope is empty and 0 otherwise.
- `isIntersecting` – determines if a set intersects a zonotope.
- `mtimes` – standard method, see Sec. 7. More details can be found in Sec. 7.1.1.
- `mptPolytope` – converts a zontope object into a mptPolytope object.
- `norm` – computes the maximum norm value of all points in a zonotope.
- `plot` – standard method, see Sec. 7. More details can be found in Sec. 7.12.
- `plotFilled` – standard method, see Sec. 7. More details can be found in Sec. 7.12.
- `plus` – standard method, see Sec. 7. More details can be found in Sec. 7.1.2.
- `polygon` – converts a two-dimensional zonotope into a polygon and returns its vertices.
- `polytope` – returns an exact polytope in halfspace representation according to [18, Theorem 2.1].
- `project` – returns a zonotope, which is the projection of the input argument onto the specified dimensions.
- `quadraticMultiplication` – given a zonotope $\mathcal{Z}$ and a discrete set of matrices $Q^{(i)} \in \mathbb{R}^{n \times n}$ for $i = 1 \dots n$, `quadraticMultiplication` computes $\{\varphi | \varphi_i = x^T Q^{(i)} x,\, x \in \mathcal{Z}\}$ as described in [19, Lemma 1].
- `quadZonotope` – converts a zonotope to a `quadZonotope` object.
- `randPoint` – generates a random point within a zonotope.
- `randPointExtreme` – generates a random extreme point of a zonotope.
- `reduce` – returns an over-approximating zonotope with fewer generators as detailed in Sec. 7.1.3.
- `split` – splits a zonotope into two or more zonotopes that enclose the original zonotope. More details can be found in Sec. 7.1.4.
- `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.

- `vertices` – returns a `vertices` object including all vertices of the zonotope (Warning: high computational complexity).
- `volume` – computes the volume of a zonotope according to [20, p.40].
- `zonotope` – constructor of the class.

7.1.1 Method `mtimes`

Table 2 lists the classes that can be multiplied with a zonotope. Please note that the order plays a role and that the zonotope has to be on the right side of the `'*'` operator.

Table 2: Classes that can be multiplied with a zonotope.

class	reference	literature
MATLAB matrix	-	-
`interval`	Sec. 7.7	[18, Theorem 3.3]
`intervalMatrix`	Sec. 8.3	[18, Theorem 3.3]
`matZonotope`	Sec. 8.2	[21, Sec. 4.4.1]

7.1.2 Method `plus`

Table 3 lists the classes that can be added to a zonotope. Unlike with multiplication, the zonotope can be on both sides of the `'+'` operator.

Table 3: Classes that can be added to a zonotope.

class	reference	literature
MATLAB vector	-	-
`zonotope`	Sec. 7.1	[18, Equation 2.1]

7.1.3 Method `reduce`

The zonotope reduction returns an over-approximating zonotope with fewer generators as described in [18, Proposition 2.5]. Table 4 lists some of the implemented reduction techniques. The standard reduction technique is `'girard'`.

Table 4: Reduction techniques for zonotopes.

technique	primary use	literature
`cluster`	Reduction to low order by clustering generators	[2, Sec. III.B]
`combastel`	Reduction of high to medium order	[22, Sec. 3.2]
`constOpt`	Reduction to low order by optimization	[2, Sec. III.D]
`girard`	Reduction of high to medium order	[17, Sec. .4]
`methA`	Reduction to low order by volume minimization (A)	Meth. A, [18, Sec. 2.5.5]
`methB`	Reduction to low order by volume minimization (B)	Meth. B, [18, Sec. 2.5.5]
`methC`	Reduction to low order by volume minimization (C)	Meth. C, [18, Sec. 2.5.5]
`pca`	Reduction of high to medium order using PCA	[2, Sec. III.A]

7.1.4 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 Table 5 can be selected to optimize the split performance.

Table 5: Split techniques for zonotopes.

split technique	**comment**	**literature**
`splitOneGen`	splits one generator	[18, Proposition 3.8]
`directionSplit`	splits all generators in one direction	—
`directionSplitBundle`	exact split using zonotope bundles	[23, Section V.A]
`halfspaceSplit`	split along a given halfspace	—

7.1.5 Zonotope Example

The following MATLAB code demonstrates some of the introduced methods:

```
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. 3 and the ones generated in lines 18-19 are shown in Fig. 4.

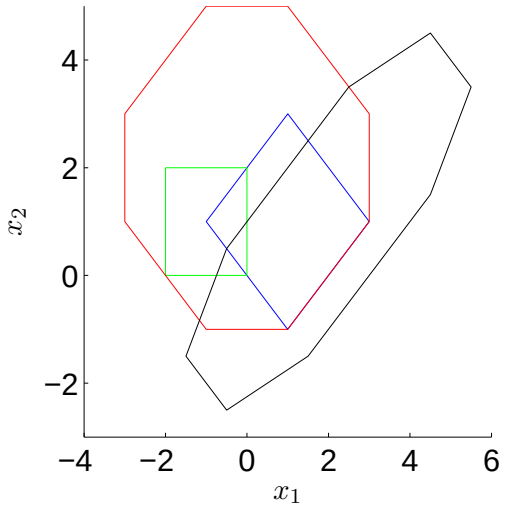


Figure 3: Zonotopes generated in lines 9-12 of the zonotope example in Sec. 7.1.5.

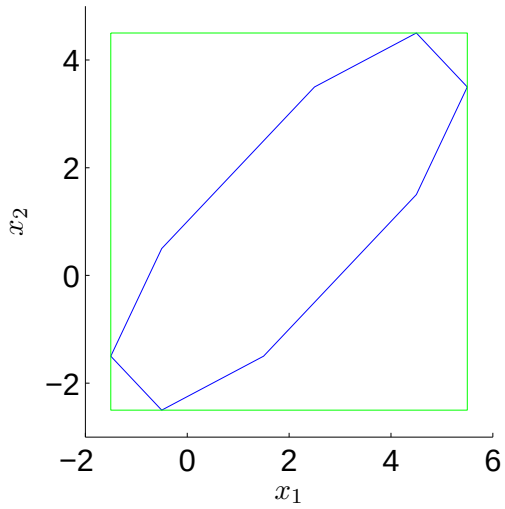


Figure 4: Sets generated in lines 18-19 of the zonotope example in Sec. 7.1.5.

7.2 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 [23]. Given a finite set of zonotopes $\mathcal{Z}_i$, a zonotope bundle is $\mathcal{Z}^{\cap} = \bigcap_{i=1}^{s} \mathcal{Z}_i$, i.e., the intersection of 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 $\mathcal{Z}^{\cap} = \{\mathcal{Z}_1, \ldots, \mathcal{Z}_s\}^{\cap}$.

We support the following methods for zonotope bundles:

- `and` – returns the intersection with a zonotope bundle or a zonotope.
- `cartesianProduct` – returns the Cartesian product of a zonotope bundle with a zonotope.
- `conZonotope` – convert a zonotope bundle into a constrained zonotope.
- `display` – standard method, see Sec. 7.
- `enclose` – generates a zonotope bundle that encloses two zonotopes bundles of equal dimension according to [23, Proposition 5].

- `encloseTight` – generates a zonotope bundle that encloses two zonotopes bundles in a possibly tighter way than `enclose` as outlined in [23, Sec. VI.A].
- `enclosingPolytope` – returns an over-approximating polytope in halfspace representation. For each zonotope the method `enclosingPolytope` of the class `zonotope` in Sec. 7.1 is called.
- `enlarge` – enlarges the generators of each zonotope in the bundle by a vector of factors for each dimension.
- `interval` – standard method, see Sec. 7. More details can be found in [23, Proposition 6].
- `mtimes` – standard method, see Sec. 7. More details can be found in [23, Proposition 1].
- `mptPolytope` – convert a zonotope bundle into a mptPolytope object.
- `plot` – standard method, see Sec. 7. More details can be found in Sec. 7.12.
- `plotFilled` – standard method, see Sec. 7. More details can be found in Sec. 7.12.
- `plus` – standard method, see Sec. 7. More details can be found in [23, Proposition 2].
- `polytope` – returns an exact polytope in halfspace representation. Each zonotope is converted to halfspace representation according to [18, Theorem 2.1] and later all obtained H polytopes are intersected.
- `project` – returns a zonotope bundle, which is the projection of the input argument onto the specified dimensions.
- `quadraticMultiplication` – given a zonotope bundle $\mathcal{Z}^{\cap}$ and a discrete set of matrices $Q^{(i)} \in \mathbb{R}^{n \times n}$ for $i = 1 \dots n$, `quadraticMultiplication` computes $\{\varphi | \varphi_i = x^T Q^{(i)} x, x \in \mathcal{Z}^{\cap}\}$ as described in [19, Lemma 1].
- `randPoint` – generates a random point within a zonotope bundle.
- `randPointExtreme` – generates a random extreme point of a zonotope bundle.
- `reduce` – returns an over-approximating zonotope bundle with less generators. For each zonotope the method `reduce` of the class `zonotope` in Sec. 7.1 is called.
- `reduceCombined` – reduces the order of a zonotope bundle by not reducing each zonotope separately as in `reduce`, but in a combined fashion.
- `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.
- `vertices` – returns potential vertices of a zonotope bundle (WARNING: Do not use this function for high order zonotope bundles due to high computational complexity).
- `volume` – computes the volume of a zonotope bundle by converting it to a polytope using `polytope` and using a volume computation for polytopes.

- `zonotopeBundle` – constructor of the class.

7.2.1 Zonotope Bundle Example

The following MATLAB code demonstrates some of the introduced methods:

```
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 = zonotopeBundle(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  plotFilled(Zb,[1 2],[.675 .675 .675],'EdgeColor','none'); % plot Zb in gray
```

This produces the workspace output

```
vol =

    1.0000
```

The plot generated in lines 7-9 is shown in Fig. 5.

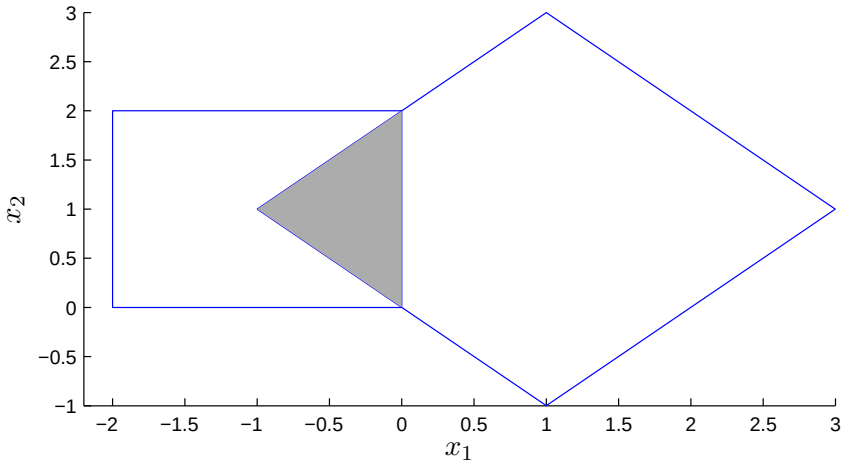


Figure 5: Sets generated in lines 7-9 of the zonotope bundle example in Sec. 7.2.1.

7.3 Polynomial Zonotopes

Zonotopes are a very efficient representation for reachability analysis of linear systems [17] and of nonlinear systems that can be well abstracted by linear differential inclusions [18]. However, more advanced techniques, such as in [24], abstract more accurately to nonlinear difference inclusions. As a consequence, linear maps of reachable sets are replaced by nonlinear maps. Zonotopes are not closed under nonlinear maps and are not particularly good at over-approximating them. For this reason, polynomial zonotopes are introduced in [24]. Polynomial zonotopes are a new non-convex set representation and can be efficiently stored and manipulated. The new representation shares many similarities with Taylor models [25] (as briefly discussed later) and is a generalization of zonotopes. Please note that a zonotope cannot be represented by a Taylor model.

Given a *starting point* $c \in \mathbb{R}^n$, multi-indexed *generators* $f^{([i],j,k,\ldots,m)} \in \mathbb{R}^n$, and single-indexed *generators* $g^{(i)} \in \mathbb{R}^n$, a polynomial zonotope is defined as

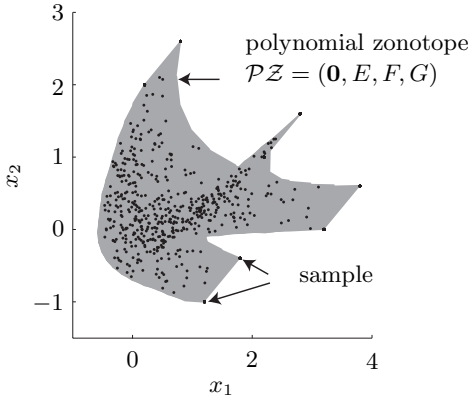
$$\mathcal{PZ} = \Big\{ c + \sum_{j=1}^{p} \beta_j f^{([1],j)} + \sum_{j=1}^{p} \sum_{k=j}^{p} \beta_j \beta_k f^{([2],j,k)} + \ldots + \sum_{j=1}^{p} \sum_{k=j}^{p} \ldots \sum_{m=l}^{p} \underbrace{\beta_j \beta_k \ldots \beta_m}_{\eta \text{ factors}} f^{([\eta],j,k,\ldots,m)} + \sum_{i=1}^{q} \gamma_i g^{(i)} \Big| \beta_i, \gamma_i \in [-1, 1] \Big\}. \tag{2}$$

The scalars β_i are called *dependent factors*, since changing their values does not only affect the multiplication with one generator, but with other generators too. On the other hand, the scalars γ_i only affect the multiplication with one generator, so they are called *independent factors*. The number of dependent factors is p, the number of independent factors is q, and the polynomial order η is the maximum power of the scalar factors β_i. The order of a polynomial zonotope is defined as the number of generators ξ divided by the dimension, which is $\rho = \frac{\xi}{n}$. For a concise notation and later derivations, we introduce the matrices

$$\begin{aligned}
E^{[i]} &= [\underbrace{f^{([i],1,1,\ldots,1)}}_{=:e^{([i],1)}} \ldots \underbrace{f^{([i],p,p,\ldots,p)}}_{=:e^{([i],p)}}] \text{ (all indices are the same value)}, \\
F^{[i]} &= [f^{([i],1,1,\ldots,1,2)} \; f^{([i],1,1,\ldots,1,3)} \; \ldots \; f^{([i],1,1,\ldots,1,p)} \\
&\quad\; f^{([i],1,1,\ldots,2,2)} \; f^{([i],1,1,\ldots,2,3)} \; \ldots \; f^{([i],1,1,\ldots,2,p)} \\
&\quad\; f^{([i],1,1,\ldots,3,3)} \ldots] \text{ (not all indices are the same value)}, \\
G &= [g^{(1)} \; \ldots \; g^{(q)}],
\end{aligned}$$

and $E = \begin{bmatrix} E^{[1]} & \ldots & E^{[\eta]} \end{bmatrix}$, $F = \begin{bmatrix} F^{[2]} & \ldots & F^{[\eta]} \end{bmatrix}$ ($F^{[i]}$ is only defined for $i \geq 2$). Note that the indices in $F^{[i]}$ are ascending due to the nested summations in (2). In short form, a polynomial zonotope is written as $\mathcal{PZ} = (c, E, F, G)$.

For a given polynomial order i, the total number of generators in $E^{[i]}$ and $F^{[i]}$ is derived using the number $\binom{p+i-1}{i}$ of combinations of the scalar factors β with replacement (i.e., the same factor can be used again). Adding the numbers for all polynomial orders and adding the number of independent generators q, results in $\xi = \sum_{i=1}^{\eta} \binom{p+i-1}{i} + q$ generators, which is in $\mathcal{O}(p^\eta)$ with respect to p. The non-convex shape of a polynomial zonotope with polynomial order 2 is shown in Fig. 6.



$$E^{[1]} = \begin{bmatrix} -1 & 0 \\ 0 & 0.5 \end{bmatrix}$$
$$E^{[2]} = \begin{bmatrix} 1 & 1 \\ 0.5 & 0.3 \end{bmatrix}$$
$$F^{[2]} = \begin{bmatrix} -0.5 \\ 1 \end{bmatrix}$$
$$G = \begin{bmatrix} 0.3 \\ 0.3 \end{bmatrix}$$

Figure 6: Over-approximative plot of a polynomial zonotope as specified in the figure. Random samples of possible values demonstrate the accuracy of the over-approximative plot.

So far, polynomial zonotopes are only implemented up to polynomial order $\eta = 2$ so that the subsequent class is called `quadZonotope` due to the quadratic polynomial order. We support the following methods for the `quadZonotope` class:

- `cartesianProduct` – returns the Cartesian product of a `quadZonotope` and a `zonotope`.
- `center` – returns the starting point c.
- `display` – standard method, see Sec. 7.
- `enclose` – generates an over-approximative `quadZonotope` that encloses two `quadZonotopes` of equal dimension by first over-approximating them by zonotopes and subsequently applying `enclose` of the `zonotope` class.
- `enclosingPolytope` – returns an over-approximating polytope in halfspace representation by first over-approximating by a `zonotope` object and subsequently applying its `enclosingPolytope` method.
- `generators` – returns the generators of a `quadZonotope`.
- `interval` – standard method, see Sec. 7. The interval hull is obtained by over-approximating the `quadZonotope` by a `zonotope` and subsequent application of its `interval` method. Other than for the `zonotope` class, the generated interval hull is not tight in the sense that it touches the `quadZonotope`.
- `intervalhullAccurate` – over-approximates a `quadZonotope` by a tighter interval hull as when applying `interval`. The procedure is based on splitting the `quadZonotope` in parts that can be more faithfully over-approximated by interval hulls. The union of the partially obtained interval hulls constitutes the result.
- `mtimes` – standard method, see Sec. 7 as stated in [23, Equation 14] for numeric matrix multiplication. As described in Sec. 7.1.1 the multiplication of interval matrices is also supported, whereas the implementation for matrix zonotopes is not yet implemented.
- `mptPolytope` – computes a mptPolytope object that encloses the quadZonotope object.
- `plot` – standard method, see Sec. 7. More details can be found in Sec. 7.12.
- `plotFilled` – standard method, see Sec. 7. More details can be found in Sec. 7.12.
- `plus` – standard method, see Sec. 7. Addition is realized for `quadZonotope` objects with MATLAB vectors, `zonotope` objects, and `quadZonotope` objects.
- `pointSet` – computes a user-defined number of random points within the `quadZonotope`.
- `pointSetExtreme` – computes a user-defined number of random points when only allowing the values $\{-1, 1\}$ for β_i, γ_i (see (2)).
- `project` – returns a `quadZonotope`, which is the projection of the input argument onto the specified dimensions.
- `quadraticMultiplication` – given a `quadZonotope` $\mathcal{Z}$ and a discrete set of matrices $Q^{(i)} \in \mathbb{R}^{n \times n}$ for $i = 1 \dots n$, `quadraticMultiplication` computes $\{\varphi | \varphi_i = x^T Q^{(i)} x,\ x \in \mathcal{Z}\}$ as described in [24, Corollary 1].
- `quadZonotope` – constructor of the class.
- `randPoint` – computes a random point within the `quadZonotope`.
- `randPointExtreme` – computes a random point when only allowing the values $\{-1, 1\}$ for β_i, γ_i (see (2)).
- `reduce` – returns an over-approximating `quadZonotope` with less generators as detailed in Sec. 7.3.1.

- `splitLongestGen` – splits the longest generator factor and returns two `quadZonotope` objects whose union encloses the original `quadZonotope` object.
- `splitOneGen` – splits one generator factor and returns two `quadZonotope` objects whose union encloses the original `quadZonotope` object.
- `zonotope` – computes an enclosing zonotope as presented in [24, Proposition 1].

7.3.1 Method `reduce`

The zonotope reduction returns an over-approximating zonotope with less generators. Table 6 lists the implemented reduction techniques.

Table 6: Reduction techniques for zonotopes.

reduction technique	**comment**	**literature**
`girard`	Only changes independent generators as for a regular zonotope	[17, Sec. 3.4]
`redistGirard`	Combines the techniques `girard` and `redistribute`	–
`redistribute`	Changes dependent and independent generators	[24, Proposition 2]

7.3.2 Polynomial Zonotope Example

The following MATLAB code demonstrates some of the introduced methods:

```
1  c = [0;0]; % starting point
2  E1 = diag([-1,0.5]); % generators of factors with identical indices
3  E2 = [1 1; 0.5 0.3]; % generators of factors with identical indices
4  F = [-0.5; 1]; % generators of factors with different indices
5  G = [0.3; 0.3]; % independent generators
6
7  qZ = quadZonotope(c,E1,E2,F,G); % instantiate quadratic zonotope
8  Z = zonotope(qZ) % over-approximate by a zonotope
9
10 figure; hold on
11 plot(Z); % plot Z
12 plotFilled(qZ,[1 2],7,[],[.6 .6 .6],'EdgeColor','none'); % plot qZ
```

This produces the workspace output

```
id: 0
dimension: 2
c:
    1.0000
    0.4000

g_i:
   -1.0000         0    0.5000    0.5000   -0.5000    0.3000
         0    0.5000    0.2500    0.1500    1.0000    0.3000
```

The plot generated in lines 11-12 is shown in Fig. 7.

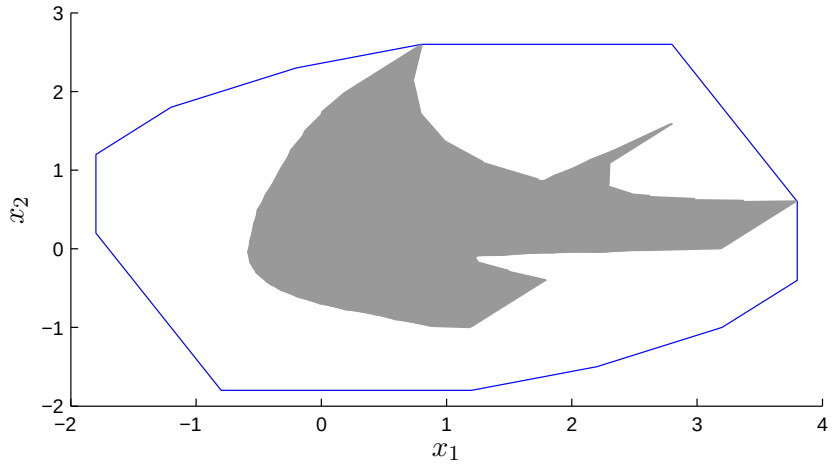


Figure 7: Sets generated in lines 11-12 of the polynomial zonotope example in Sec. 7.3.2.

7.4 Probabilistic Zonotopes

Probabilistic zonotopes have been introduced in [26] for stochastic verification. A probabilistic zonotope has the same structure as a zonotope, except that the values of some β_i in (1) are bounded by the interval $[-1, 1]$, while others are subject to a normal distribution [6]. Given pairwise independent Gaussian distributed random variables $\mathcal{N}(\mu, \Sigma)$ with expected value μ and covariance matrix Σ, one can define a Gaussian zonotope with certain mean:

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

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

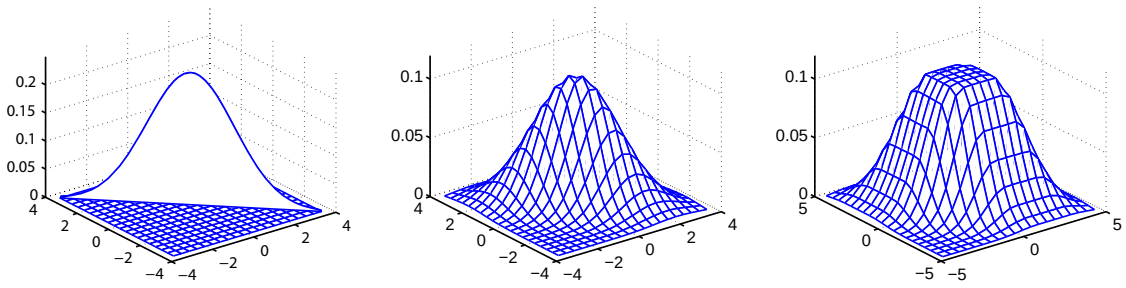
$$\mathscr{Z} := \mathcal{Z} \boxplus \mathcal{Z}_g, \quad \mathcal{Z} = (c, g^{(1\ldots p)}),\ \mathcal{Z}_g = (0, \underline{g}^{(1\ldots q)}),$$

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 Σ ($q > n$) as shown in [26, Proposition 1], one can also write $\mathscr{Z} = (c, g^{(1\ldots p)}, \Sigma)$. As $\mathscr{Z}$ 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 [27]. Analogously to a zonotope, it is shown in Fig. 8 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.

We support the following methods for probabilistic zonotopes:

- `center` – returns the center of the probabilistic zonotope.
- `display` – standard method, see Sec. 7.

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



(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 8: Construction of a probabilistic zonotope.

- `enclose` – generates a probabilistic zonotope that encloses two probabilistic zonotopes $\mathscr{Z}$, $A \otimes \mathscr{Z}$ ($A \in \mathbb{R}^{n \times n}$) of equal dimension according to [26, Section VI.A].
- `enclosingProbability` – computes values to plot the mesh of a two-dimensional projection of the enclosing probability hull.
- `max` – computes an over-approximation of the maximum on the m-sigma bound according to [26, Equation 3].
- `mean` – returns the uncertain mean of a probabilistic zonotope.
- `mtimes` – standard method, see Sec. 7 as stated in [26, Equation 4] for numeric matrix multiplication. The multiplication of interval matrices is also supported.
- `plot` – standard method, see Sec. 7. More details can be found in Sec. 7.12.
- `plus` – standard method, see Sec. 7. Addition is realized for `probZonotope` objects with MATLAB vectors, `zonotope` objects, and `probZonotope` objects as described in [26, Equation 4].
- `probReduce` – reduces the number of single Gaussian distributions to the dimension of the state space.
- `probZonotope` – constructor of the class.
- `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 [26, Section VI.C].
- `reduce` – returns an over-approximating zonotope with fewer generators. The zonotope of the uncertain mean $\mathcal{Z}$ is reduced as detailed in Sec. 7.1.3, while the order reduction of the probabilistic part is done by the method `probReduce`.
- `sigma` – returns the Σ matrix of a probabilistic zonotope.
- `zonotope` – converts a probabilistic zonotope to a common zonotope where for each generator, a m-sigma interval is taken.

7.4.1 Probabilistic Zonotope Example

The following MATLAB code demonstrates some of the introduced methods:

```
1 Z1=[10 ; 0 ]; % uncertain center
2 Z2=[0.6 1.2  ; 0.6 -1.2]; % generators with normally distributed factors
```

```
3  pZ=probZonotope(Z1,Z2,2); % probabilistic zonotope
4
5  M=[-1 -1;1 -1]*0.2; % mapping matrix
6  pZencl = enclose(pZ,M); % probabilistic enclosure of pZ and M*pZ
7
8  figure('renderer','zbuffer')
9  hold on
10 plot(pZ,'dark'); % plot pZ
11 plot(expm(M)*pZ,'light'); % plot expm(M)*pZ
12 plot(pZencl,'mesh') % plot enclosure
13
14 campos([-3,-51,1]); %set camera position
15 drawnow; % draw 3D view
```

The plot generated in lines 10-15 is shown in Fig. 9.

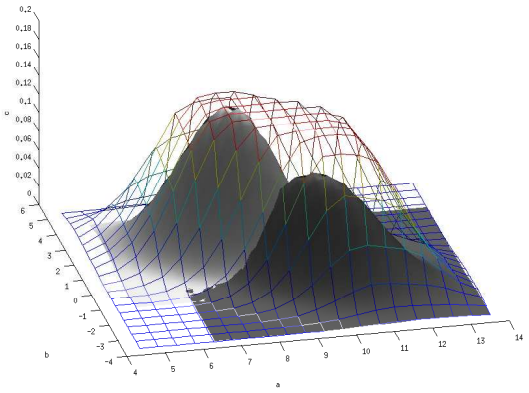


Figure 9: Sets generated in lines 10-15 of the probabilistic zonotope example in Sec. 7.4.1.

7.5 Constrained Zonotopes

An extension of zonotopes described in Sec. 7.1 are constrained zonotopes, which are introduced in [4]. A constrained zonotope is defined as a zonotope with additional equality constraints on the factors β_i:

$$\mathcal{Z}_c = \left\{ c + G\beta \middle| \|\beta\|_\infty \leq 1, A\beta = b \right\}, \tag{3}$$

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. 7.6) 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 implemented in the class `conZonotope`, which supports the following methods:

- `and` – computes the intersection of a `conZonotope` object with other set representations. More details can be found in Sec. 7.5.1.

- `boundDir` – computes an upper and lower bound for the projection of a `conZonotope` onto a certain state space direction (vector).
- `conZonotope` – constructor of the class.
- `display` – standard method, see Sec. 7.
- `enclose` – generates a `conZonotope` object that encloses two constrained zonotopes. More details can be found in Sec. 7.5.2.
- `interval` – standard method, see Sec. 7.
- `isempty` – returns 1 if a `conZonotope` object is empty and 0 otherwise.
- `mptPolytope` – converts a `conZonotope` object into a `mptPolytope` object.
- `mtimes` – standard method, see Sec. 7. More details can be found in [4].
- `plot` – standard method, see Sec. 7. More details can be found in Sec. 7.12.
- `plotFilled` – standard method, see Sec. 7. More details can be found in Sec. 7.12.
- `plotZono` – plots a two-dimensional projection of the `conZonotope` object together with the corresponding zonotope.
- `plus` – standard method, see Sec. 7. More details can be found in Sec. 7.5.3.
- `project` – returns a `conZonotope` object, which is the projection of the input argument onto the specified dimensions.
- `reduce` – returns an over-approximating `conZonotope` object with fewer generators and/or constraints as detailed in Sec. 7.5.4.
- `rescale` – prune the domain of the zonotope factors β_i by adequate adaption of the zonotope generators. More details can be found in [4].
- `vertices` – returns a `vertices` object including all vertices of the `conZonotope` object (Warning: high computational complexity).
- `zonotope` – returns a `zonotope` object that over-approximates the `conZonotope` object.

7.5.1 Method `and`

Table 7 lists the classes that can be intersected with a `conZonotope` object. Please note that the order plays a role and that the `conZonotope` object has to be on the left side of the `'&'` operator.

Table 7: Classes that can be intersected with a `conZonotope` object.

class	**reference**	**literature**
`conZonotope`	Sec. 7.5	[4, Proposition 1]
`zonotope`	Sec. 7.1	-
`interval`	Sec. 7.7	-
`mptPolytope`	Sec. 7.6	-
`halfspace`	-	-
`constrainedHyperplane`	-	-

7.5.2 Method `enclose`

The method `enclose` can only be applied if one constrained zonotope $\mathcal{Z}_{c,2}$ represents a linear transformation of the other constrained zonotope $\mathcal{Z}_{c,1}$:

$$\mathcal{Z}_{c,2} = T \cdot \mathcal{Z}_{c,1} + t, \tag{4}$$

where $T \in \mathbb{R}^{m \times n}$ is a transformation matrix and n is the dimension of the state space. Table 8 lists the classes that are valid values for the variable t in (4). The reason for this requirement is that with the restriction implied by (4) it is possible to calculate a very tight enclosure of the `conZonotope` objects $\mathcal{Z}_{c,1}$ and $\mathcal{Z}_{c,2}$. In addition, the main application of the `enclose` function in CORA is the calculation of the convex hull during reachability analysis, where the restriction formulated in (4) is always fulfilled.

Table 8: Classes that represent valid values for the vector t in (4).

class	reference	literature
MATLAB vector	-	-
`zonotope`	Sec. 7.1	-
`interval`	Sec. 7.7	-

7.5.3 Method `plus`

Table 9 lists the classes that can be added to a `conZonotope` object. Unlike with intersection, the `conZonotope` object can be on both sides of the `'+'` operator.

Table 9: Classes that can be added to a `conZonotope` object.

class	reference	literature
MATLAB vector	-	-
`conZonotope`	Sec. 7.5	[4, Proposition 1]
`zonotope`	Sec. 7.1	-
`interval`	Sec. 7.7	-

7.5.4 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 [4, Section 4.2], and reduction of the *degrees-of-freedom order* o_c [4, Section 4.3].

7.5.5 Constrained Zonotope Example

The following MATLAB code demonstrates some of the introduced methods:

```
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. 10. Fig. 11 displays a visualization of the constraints for the `conZonotope` object that is shown in Fig. 10.

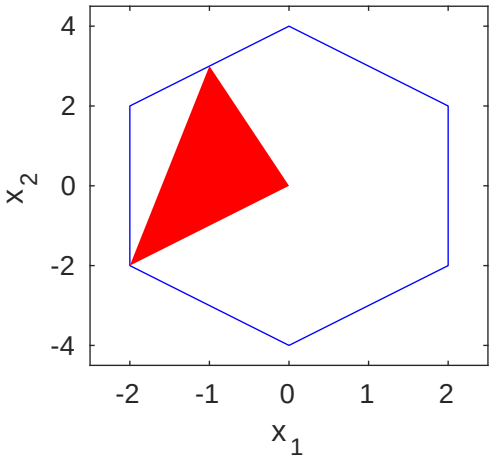


Figure 10: Zonotope (blue) and the corresponding constrained zonotope (red) generated in the constrained zonotope example in Sec. 7.5.5

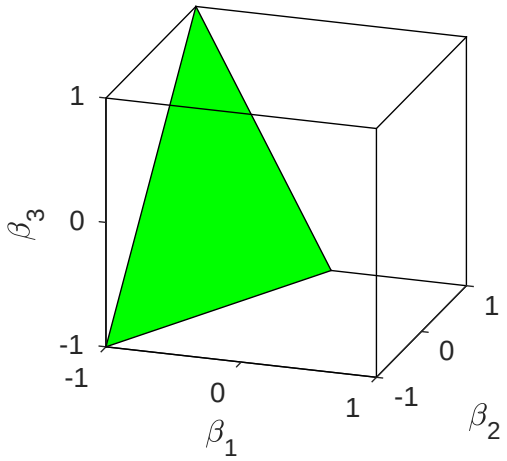


Figure 11: Visualization of the constraints for the `conZonotope` object generated in the constrained zonotope example in Sec. 7.5.5.

7.6 MPT Polytopes

There exist two representations for polytopes: The halfspace representation (H-representation) and the vertex representation (V-representation). 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 $\mathcal{S}$, which is given by $\mathcal{S} := \{x | c^T x = d\}, c \in \mathbb{R}^n, d \in \mathbb{R}$. The vector c is the normal vector of the hyperplane and d the scalar product of any point on the hyperplane with the normal vector.

From this follows that the corresponding halfspace is $\mathcal{H} := \{x | c^T x \leq d\}$. As the convex polytope $\mathcal{P}$ is the nonempty intersection of q halfspaces, q inequalities have to be fulfilled simultaneously.

H-Representation of a Polytope A convex polytope $\mathcal{P}$ is the bounded intersection of q halfspaces:

$$\mathcal{P} = \Big\{ x \in \mathbb{R}^n \big| C\, x \leq d \Big\}, \quad C \in \mathbb{R}^{q \times n}, d \in \mathbb{R}^q.$$

When the intersection is unbounded, one obtains a polyhedron [28].

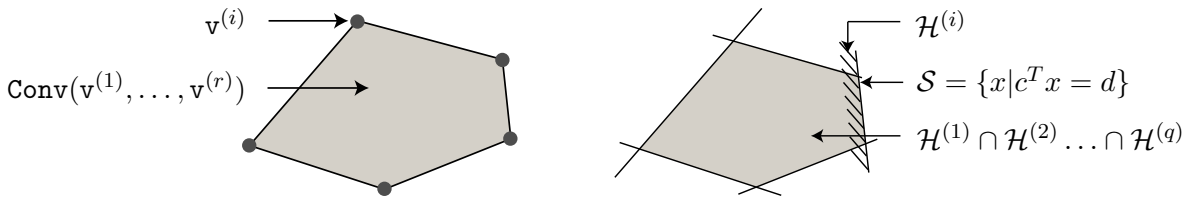
A polytope with vertex representation is defined as the convex hull of a finite set of points in the n-dimensional Euclidean space. The points are also referred to as vertices and are denoted by $\mathtt{v}^{(i)} \in \mathbb{R}^n$. A convex hull of a finite set of r points $\mathtt{v}^{(i)} \in \mathbb{R}^n$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

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

Given the convex hull operator `Conv()`, a convex and bounded polytope can be defined in vertex representation as follows:

V-Representation of a Polytope For r vertices $\mathtt{v}^{(i)} \in \mathbb{R}^n$, a convex polytope $\mathcal{P}$ is the set $\mathcal{P} = \mathtt{Conv}(\mathtt{v}^{(1)}, \ldots, \mathtt{v}^{(r)})$.

The halfspace and the vertex representation are illustrated in Fig. 12. Algorithms that convert from H- to V-representation and vice versa are presented in [29].



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

Figure 12: Possible representations of a polytope.

The class `mptPolytope` is a wrapper class that interfaces with the MATLAB toolbox *Multi-Parametric Toolbox* (MPT) for the following methods:

- `and` – computes the intersection of two `mptPolytopes`.
- `conZonotope` – converts a `mptPolytope` object into a constrained zonotope.
- `dimension` – returns the dimension of an `mptPolytope`.
- `display` – standard method, see Sec. 7.
- `enclose` – computes the convex hull of two `mptPolytopes`.
- `in` – determines if a `zonotope` is enclosed by a `mptPolytope`.
- `interval` – encloses a `mptPolytope` by intervals of INTLAB.
- `interval` – encloses a `mptPolytope` by an `interval`.
- `iscontained` – returns if a `mptPolytope` is contained in another `mptPolytope`.
- `isempty` – returns 1 if a `mptPolytope` is empty and 0 otherwise.

- `le` – overloads the '¡=' operator; returns 1 if one polytopes is equal or enclosed by the other one and 0 otherwise.
- `mldivide` – computes the set difference of two `mptPolytopes`.
- `mldivide` – computes the set difference $P_1 \setminus P_2$ such that P_2 is subtracted from P_1.
- `mptPolytope` – constructor of the class.
- `minus` – overloaded '-' operator for the subtraction of a vector from an `mptPolytope` or the Minkowski difference between two `mptPolytope` objects.
- `mtimes` – standard method, see Sec. 7 for numeric and interval matrix multiplication.
- `or` – overloaded '—' operator to compute the union of two `mptPolytopes`.
- `plot` – standard method, see Sec. 7. More details can be found in Sec. 7.12.
- `plotFilled` – standard method, see Sec. 7. More details can be found in Sec. 7.12.
- `project` – projects a `mptPolytope` onto a set of dimensions.
- `plus` – standard method, see Sec. 7 for numeric vectors and `mptPolytope` objects.
- `vertices` – returns a `vertices` object including all vertices of the polytope.
- `volume` – computes the volume of a polytope.

7.6.1 MPT Polytope Example

The following MATLAB code demonstrates some of the introduced methods:

```
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
plotFilled(P4,[1 2],[.6 .6 .6],'EdgeColor','none'); % plot P4
```

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: 2.8284
```

```
[ 0.70711    -0.70711]       [1.4142]
[       0          -1]       [     1]
[-0.70711    -0.70711]       [1.4142]
[      -1           0]       [     3]
[-0.70711     0.70711] x <= [4.2426]
[       0           1]       [     5]
[ 0.70711     0.70711]       [4.2426]
[       1           0]       [     3]

V:
         0   -1.0000         0
         0    1.0000    2.0000
```

The plot generated in lines 13-16 is shown in Fig. 13.

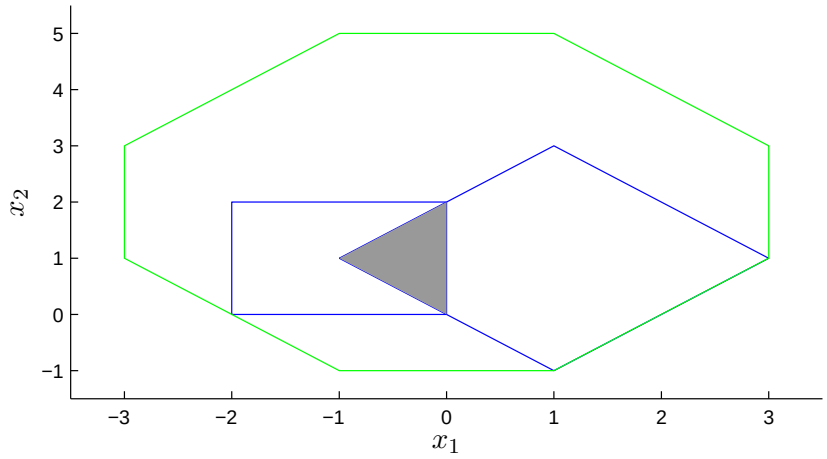


Figure 13: Sets generated in lines 13-16 of the MPT polytope example in Sec. 7.6.1.

7.7 Intervals

A real-valued interval $[x] = [\underline{x}, \overline{x}] = \{x \in \mathbb{R} | \underline{x} \leq x \leq \overline{x}\}$ is a connected subset of $\mathbb{R}$ and can be specified by a left bound $\underline{x} \in \mathbb{R}$ and right bound $\overline{x} \in \mathbb{R}$, where $\underline{x} \leq \overline{x}$. A detailed description of how intervals are treated in CORA can be found in [7]. 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

- `abs` – returns the absolute value as defined in [7, Eq. (10)].
- `acos` – arccos(·) function as defined in [7, Eq. (6)].
- `acosh` – arccosh(·) function as defined in [7, Eq. (8)].
- `and` – computes the intersection of two `intervals` as defined in [7, Eq. (1)].
- `asin` – arcsin(·) function as defined in [7, Eq. (6)].
- `asinh` – arcsinh(·) function as defined in [7, Eq. (8)].
- `atan` – arctan(·) function as defined in [7, Eq. (6)].

- `atanh` – arctanh(·) function as defined in [7, Eq. (8)].
- `cos` – cos(·) function as defined in [7, Eq. (13)].
- `cosh` – cosh(·) function as defined in [7, Eq. (7)].
- `ctranspose` – overloaded ' ' operator for single operand to transpose a matrix.
- `eq` – overloads the '==' operator to check if both intervals are equal.
- `exp` – exponential function as defined in [7, Eq. (4)].
- `le` – overloads `<=` operator: Is one interval equal or the subset of another interval?
- `log` – natural logarithm function as defined in [7, Eq. (5)].
- `lt` – overloads `<` operator: Is one interval equal or the subset of another interval?
- `minus` – overloaded '-' operator, see [7, Eq. (2)].
- `mpower` – overloaded '^' operator (power), see [7, Eq. (9)].
- `mrdivide` – overloaded '/' operator (division), see [7, Eq. (3)].
- `mtimes` – overloaded '*' operator (multiplication), see [7, Eq. (2)] for scalars and [7, Eq. (16)] for matrices.
- `ne` – overloaded ' =' operator.
- `plus` – overloaded '+' operator (addition), see [7, Eq. (2)] for scalars and [7, Eq. (17)] for matrices.
- `power` – overloaded '.^' operator for intervals (power), see [7, Eq. (9)].
- `prod` – product of array elements.
- `rdivide` – overloads the './' operator: provides elementwise division of two matrices.
- `sin` – sin(·) function as defined in [7, Eq. (12)].
- `sinh` – sinh(·) function as defined in [7, Eq. (7)].
- `sqrt` – $\sqrt{(\cdot)}$ function as defined in [7, Eq. (5)].
- `tan` – tan(·) function as defined in [7, Eq. (14)].
- `tanh` – tanh(·) function as defined in [7, 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.
- `display` – standard method, see Sec. 7.
- `enclosingRadius` – computes radius of enclosing hyperball of an interval.
- `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]`.
- `hull` – returns the union of two intervals.
- `in` – determines if elements of a zonotope are in an interval.
- `infimum` – returns the infimum of an interval.
- `interval` – constructor of the class.
- `isempty` – returns 1 if an interval is empty and 0 otherwise.
- `isIntersecting` – determines if a set intersects 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.
- `mid` – returns the center of an interval.
- `plot` – standard method, see Sec. 7. More details can be found in Sec. 7.12.
- `plotFilled` – standard method, see Sec. 7. More details can be found in Sec. 7.12.
- `polytope` – converts an interval object to a polytope.
- `rad` – returns the radius of 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.
- `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]`.
- `vertices` – returns a `vertices` object including all vertices.
- `volume` – computes the volume of an interval.
- `zonotope` – converts an `interval` object to a `zonotope` object.

7.7.1 Interval Example

The following MATLAB code demonstrates some of the introduced methods:

```
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 = mid(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 plotFilled(I3,[1 2],[.6 .6 .6],'EdgeColor','none'); % 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. 14.

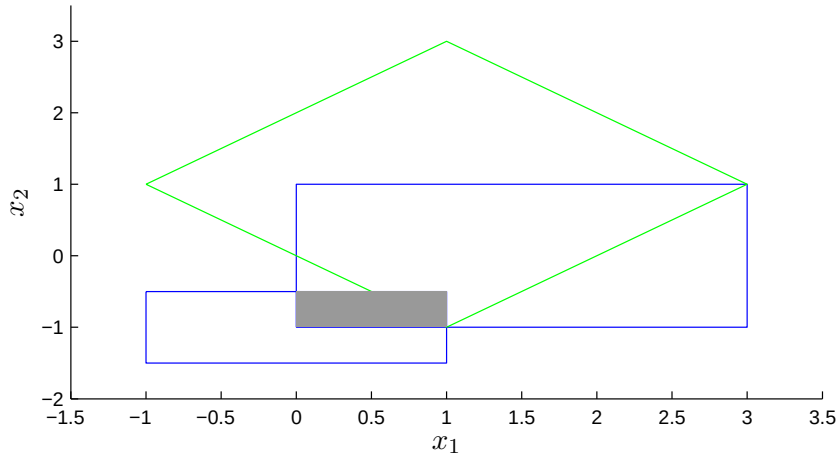


Figure 14: Sets generated in lines 11-14 of the interval example in Sec. 7.7.1.

7.8 Taylor Models

Taylor models [30–33] can be used to obtain rigorous bounds of functions that are often tighter than the ones obtained by interval arithmetic. To define Taylor models, we first introduce an n-dimensional interval $[x] := [\underline{x}, \overline{x}], \forall i : \quad \underline{x}_i \leq \overline{x}_i, \underline{x}, \overline{x} \in \mathbb{R}^n$. Let us next introduce the multi-index

set (see Sec. 3 in [34])

$$\mathcal{L}^q = \left\{ (l_1, l_2, \ldots, l_n) \middle| l_i \in \mathbb{N}, \sum_{i=1}^{n} l_i \leq q \right\}.$$

We define $P^q(x - x_0)$ as the q-th order Taylor polynomial of $f(x)$ around x_0 ($x, x_0 \in \mathbb{R}^n$):

$$P^q(x - x_0) = \sum_{l \in \mathcal{L}^q} \frac{(x_1 - x_{0,1})^{l_1} \ldots (x_n - x_{0,n})^{l_n}}{l_1! \ldots l_n!} \left(\frac{\partial^{l_1 + \ldots + l_n} f(x)}{\partial x_1^{l_1} \ldots \partial x_n^{l_n}} \right) \Bigg|_{x=x0}. \tag{5}$$

Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a function that is $(q+1)$ times continuously differentiable in an open set containing the n-dimensional interval $[x]$. Given $P^q(x - x_0)$ as the q-th order Taylor polynomial of $f(x)$ around $x_0 \in [x]$, we choose an n-dimensional interval $[I]$ such that

$$\forall x \in [x]: \quad f(x) \in P^q(x - x_0) + [I]. \tag{6}$$

The pair $T = (P^q(x - x_0), I)$ is called an q-th order Taylor model of $f(x)$ around x_0 (see Def. 1 in [32]). From now on we use the shorthand notation (P, I), and we omit q, x, and x_0 when it is self-evident. Further information can be found in [31, Sec. 2]. An illustration of a fourth-order Taylor model is shown in Fig. 15 for $r = \cos(x)$ and the range $[x] = [-\pi/3, \pi/2]$.

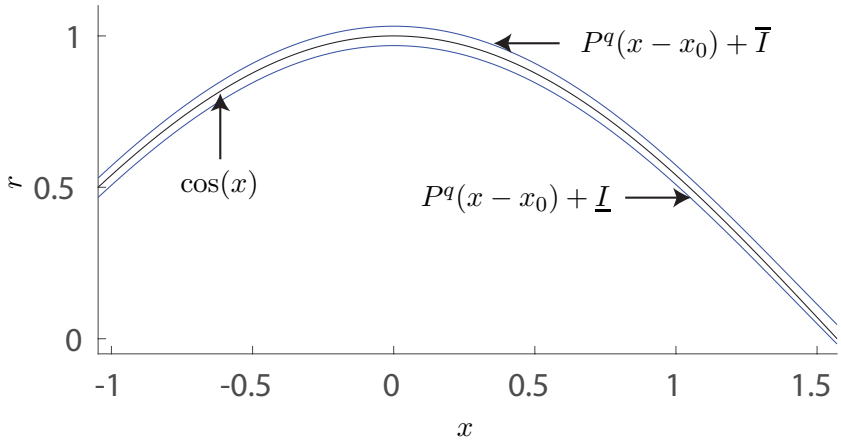


Figure 15: Fourth-order Taylor model for $\cos(x)$ and $[x] = [-\pi/3, \pi/2]$.

A detailed description of how Taylor models are treated in CORA can be found in [3]. In contrast to interval arithmetic and affine arithmetic, Taylor models do not directly provide a range of possible values. The bounds of a Taylor model T with a polynomial P and interval $[I]$ can be over-approximated as

$$B(T) = B(P) + [I]$$

using

$$B(P^q(x - x_0)) = [\min_{x \in [x]} P^q(x - x_0), \max_{x \in [x]} P^q(x - x_0)]. \tag{7}$$

Several approaches to obtain $B(P)$ exist, out of which *interval arithmetic*, *branch and bound*, and the *LDB/QFB algorithm* are currently implemented in CORA.

7.8.1 Creating Taylor Models

Taylor models are implemented in CORA by the 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.

```
a1 = interval(-1, 2); % generate a scalar interval [-1,2]
b1 = taylm(a1, 6); % generate a scalar Taylor model of order 6
a2 = interval(2, 3); % generate a scalar interval [2,3]
b2 = taylm(a2, 6); % generate a scalar Taylor model of order 6
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.

```
a = interval([-1;2], [2;3]); % generate an interval vector [[-1,2]; [2,3]]
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]
```

7.8.2 List of Functions of the Class `taylm`

In this subsection we list the functions realized in CORA. Since CORA is implemented in MATLAB, the function names are chosen such that they overload the built-in MATLAB functions. 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 [3, Eq. (31)].
- `asin` – arcsin(·) function as defined in [3, Eq. (30)].
- `atan` – arctan(·) function as defined in [3, Eq. (32)].
- `cos` – cos(·) function as defined in [3, Eq. (25)].
- `cosh` – cosh(·)function as defined in [3, Eq. (28)].
- `det` – determinant of a Taylor model matrix.
- `exp` – exponential function as defined in [3, Eq. (21)].
- `interval` – various implementations of the bound operator $B(\cdot)$ as presented in [3, Sec. 2.3].
- `log` – natural logarithm function as defined in [3, Eq. (22)].
- `minus` – overloaded '-' operator, see [3, Eq. (7)].
- `mpower` – overloaded '^' operator (power).
- `mrdivide` – overloaded '/' operator (division), see [3, Eq. (9)].
- `mtimes` – overloaded '*' operator (multiplication), see [3, Eq. (8)] for scalars and [3, Sec. 2.4] for matrices.
- `plus` – overloaded '+' operator (addition), see [3, Eq. (6)] for scalars and [3, Sec. 2.4] for matrices.
- `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 [3, Eq. (24)].
- `sinh` – sinh(·) function as defined in [3, Eq. (27)].
- `sqrt` – $\sqrt{(\cdot)}$ function as defined in [3, Eq. (23)].
- `tan` – tan(·) function as defined in [3, Eq. (26)].
- `tanh` – tanh(·) function as defined in [3, 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

- `display` – displays the values of a `taylm` object in the MATLAB workspace.
- `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.
- `optBnb` – implementation of the branch and bound algorithm as presented in [3, Sec. 2.3.2].
- `optBnbAdv` – implementation of the advanced branch and bound algorithm as presented in [3, Sec. 2.3.2].
- `optLinQuad` – implementation of the algorithm based on LDB and QFB as presented in [3, Sec. 2.3.3].
- `horzcat` – overloads the operator for horizontal concatenation, e.g., `a = [b, c, d]`.
- `set` – set the additional class parameters (see [3, 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.
- `taylm` – constructor of the `taylm` class.
- `vertcat` – overloads the operator for vertical concatenation, e.g., `a = [b; c; d]`.

7.8.3 Additional Parameters for the Class `taylm`

CORA's Taylor model implementation contains some additional parameters which can be modified by the user:

- **max_order:** Maximum polynomial degree of the monomials in the polynomial part of the Taylor model. Monomials with a degree larger than **max_order** are bounded with the bounding operator $B(\cdot)$ and added to the interval remainder. Further, $q =$ **max_order** is used for the implementation of the formulas listed in [3, Appendix A].
- **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 bounded with the bounding operator $B(\cdot)$ and added to the interval remainder.
- **eps:** Termination tolerance ϵ for the branch and bound algorithm from [3, Sec. 2.3.2] and for the algorithm based on the Linear Dominated Bounder and the Quadratic Fast Bounder from [3, Sec. 2.3.3].

These parameters are stored as properties of the class `taylm`. In the functions `plus`, `minus`, and `times`, two Taylor models are combined to one resulting Taylor model object using the rules

$$\begin{aligned} \text{max\_order}_{new} &= \max(\text{max\_order}_1, \text{max\_order}_2), \\ \text{tolerance}_{new} &= \min(\text{tolerance}_1, \text{tolerance}_2), \\ \text{eps}_{new} &= \min(\text{eps}_1, \text{eps}_2), \end{aligned} \tag{8}$$

where the subscript *new* refers to the resulting object, and 1 and 2 to the initial objects.

7.8.4 Talyor Models for Reachability Analysis

During reachabilty analysis for nonlinear systems (see Sec. 9.4), the set of abstraction errors $\mathcal{L}$ is obtained by computing upper and lower bounds for the Lagrange remainder function of the Taylor series. This is a classical range bounding problem, and therefore Taylor models can be applied to solve it. Since Taylor models often lead to tighter bounds compared to interval arithmetic, the expection is that the usage of Taylor models leads to a tighter over-approximation of the set of abstraction errors, which then results in a tighter over-approximation of the reachable set. The following algorithm settings enable the evaluation of the set of abstraction errors with Taylor models:

- `options.lagrangeRem.method`: Method that is used to compute the bounds for the set of abstraction errors. The available methods are 'interval' (interval arithmetic), 'taylorModel' or 'zoo'. The default value is 'interval'.
- `options.lagrangeRem.maxOrder`: Maximum polynomial degree of the monomials in the polynomial part of the Taylor model (see Sec. 7.8.3).
- `options.lagrangeRem.optMethod`: Method used to calculate the bounds of the Talyor model objects. The available methods are 'int' (interval arithmetic), 'bnb' (branch and bound algorithm), 'bnbAdv' (branch and bound with Taylor model re-expansion) and 'linQuad' (optimization with Linear Dominated Bounder and Quadratic Fast Bounder)
- `options.lagrangeRem.tolerance`: Minimum absolute value of the monomial coefficients in the polynomial part of the Taylor model (see Sec. 7.8.3).
- `options.lagrangeRem.eps`: Termination tolerance ϵ for the branch and bound algorithm and the algorithm based on the Linear Dominated Bounder and the Quadratic Fast Bounder (see Sec. 7.8.3).

A list that summarizes all available algorithm settings is provided later on in Section 12.

7.8.5 Taylor Model Example

This section presents the results of several examples evaluated in CORA:

```
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
```

```
21  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]
```

7.9 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., [35]):

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

The possible values of $\hat{x}$ lie within a zonotope [36]. In contrast to Taylor models, it is possible that $p > n$ so that affine forms are not a special case of Taylor models. This is the same reason

why polynomial zonotopes are different from Taylor models, since polynomial zonotopes are a generalization of zonotopes [24].

For the CORA class `affine` we only consider intervals as inputs and outputs, and therefore realize affine arithmetic as Taylor models of first order. The class 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.

The following code example demonstrates the usage of the class `affine`:

```
% create affine object
int = interval(-1,1);
aff = affine(int);

% 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]
```

7.10 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 using one accurate method. This fact is nicely demonstrated by the numerical examples shown in [3] and by the code example at the end of this section. To facilitate mixing different range bounding techniques, we created the class `zoo` in which one can specify the methods to be combined.

The following list summerizes all range bounding techniques that are available for class `zoo`:

- `interval` – Interval arithmetic (see Sec. 7.7).
- `affine(int)` – Affine arithmetic; the bounds of the affine objects are calculated with interval arithmetic (see Sec. 7.9).
- `affine(bnb)` – Affine arithmetic; the bounds of the affine objects are calculated with the branch and bound algorithm (see Sec. 7.9).
- `affine(bnbAdv)` – Affine arithmetic; the bounds of the affine objects are calculated with the advanced branch and bound algorithm (see Sec. 7.9).
- `affine(linQuad)` – Taylor models; 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. 7.9).

- `taylm(int)` – Taylor models; the bounds of the Taylor models are calculated with interval arithmetic (see Sec. 7.8).
- `taylm(bnb)` – Taylor models; the bounds of the Taylor models are calculated with the branch and bound algorithm (see Sec. 7.8).
- `taylm(bnbAdv)` – Taylor models; the bounds of the Taylor models are calculated with the advanced branch and bound algorithm (see Sec. 7.8).
- `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. 7.8).

In addition to the range bounding techniques, the additional Taylor model parameters described in Sec. 7.8.3 as well as the names of the variables can be specified as additional parameters for the class `zoo`. The bounds of a zoo object can be computed with the `interval` operator, which intersects the intervals obtained by all specified techniques. All functions that are implemented for class `taylm` are also available for the class `zoo`.

Similar to Taylor models, `zoo` objects can be used to calculate tight bounds of the Lagrange remainder function during reachability analysis for nonlinear systems (see Sec. 7.8.4). When using `zoo` objects, the desired range bounding techniques have to be specified as a cell-array of strings in `options.lagrangeRem.zooMethods`. A list that summarizes all algorithm settings is provided later on in Sec. 12.

The following code example demonstrates the usage of the class `zoo`:

```
% create zoo object
int = interval(-1,1);
methods = {'interval','taylm(int)'};
maxOrder = 3;
z = zoo(int,methods,maxOrder);

% create taylor model object (for comparison)
maxOrder = 10;
tay = taylm(int,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]
```

7.11 Vertices

The `vertices` class performs operations on a set of vertices, such as enclosing them by a parallelotope. The following methods are implemented:

- `collect` – collects cell arrays (MATLAB-specific container) of vertices.

- `display` – standard method, see Sec. 7.
- `interval` – encloses all vertices by an `interval`.
- `isempty` – returns 1 if there is no vertex and 0 otherwise.
- `mtimes` – standard method, see Sec. 7 for numeric matrix multiplication.
- `parallelotope` – computes a parallelotope in generator representation based on a coordinate transformation in which the transformed vertices are enclosed by an interval hull.
- `plot` – standard method, see Sec. 7. More details can be found in Sec. 7.12.
- `plot3d` – plots the convex hull of vertices in 3D.
- `plotFilled` – standard method, see Sec. 7. More details can be found in Sec. 7.12.
- `plus` – standard method, see Sec. 7. Addition is only realized for `vertices` objects with MATLAB vectors.
- `polygon` – computes ordered lists of vertices, defining a polygon; only the first two coordinates are considered.
- `subsref` – overloads the operator that selects elements of a vertex matrix `V`, e.g., `value=V(1,2)`, where the element of the first row and second column is read.
- `vertices` – constructor of the class.
- `zonotope` – computes a zonotope that encloses all vertices according to [37, Section 3].

7.11.1 Vertices Example

The following MATLAB code demonstrates some of the introduced methods:

```
1  Z1 = zonotope([1 1 1; 1 -1 1]); % create zonotope Z1
2  V1 = vertices(Z1); % compute vertices of Z1
3  A = [0.5 1; 1 0.5]; % numerical matrix A
4
5  V2{1} = A*V1; % linear map of vertices
6  V2{2} = V2{1} + [1; 0]; % translation of vertices
7  V3 = collect(V2{1},V2); % collect vertices of cell array V2
8  Zencl = zonotope(V3); % obtain parallelotope containing all vertices
9
10 figure
11 hold on
12 plot(V2{1},'k+'); % plot V2{1}
13 plot(V2{2},'ko'); % plot V2{2}
14 plot(Zencl); % plot Zencl
```

The plot generated in lines 11-14 is shown in Fig. 16.

7.12 Plotting of Sets

Plotting of reachable sets is performed by first projecting the set onto two dimensions. Those dimensions can be two states for plots in state space, or a state and a time interval for plots involving a time axis. A selection of plot types is presented in Fig. 17 for two zonotopes using the standard MATLAB `LineSpec`, `ColorSpec`, and `Patch` settings. The command `plot` only

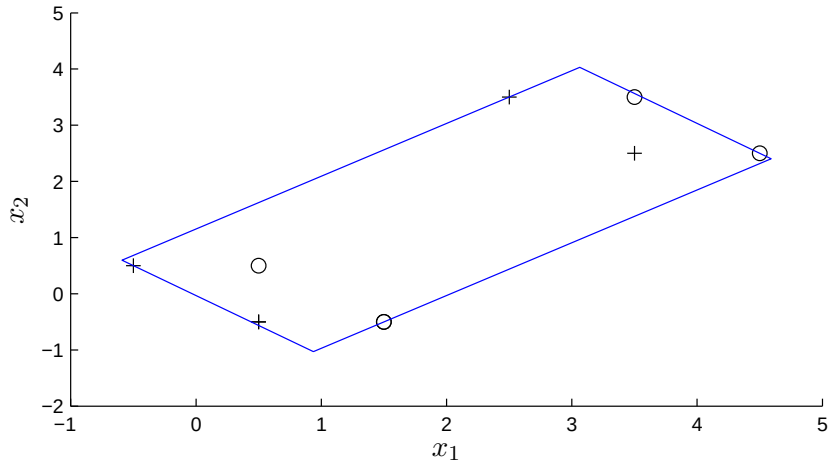


Figure 16: Sets generated in lines 11-14 of the vertices example in Sec. 7.11.1.

plots the edge, while `plotFilled` also fills the sets. The corresponding standard MATLAB functions are `plot` and `fill`, respectively.

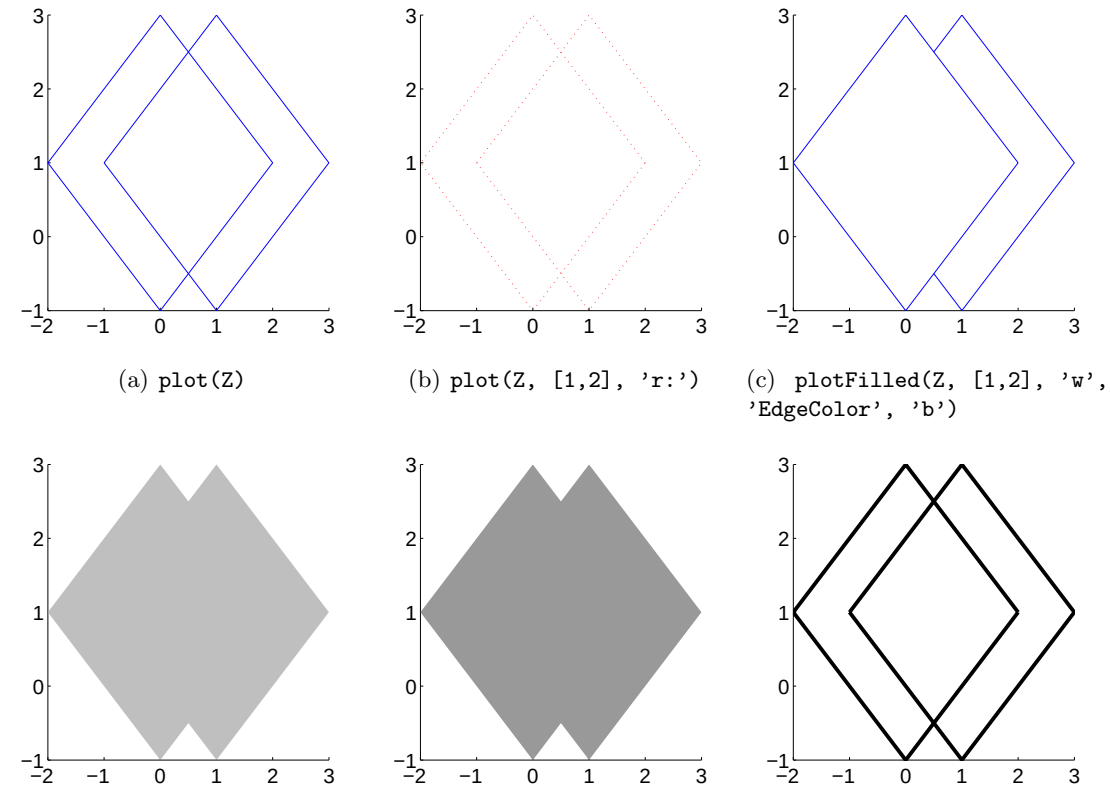
8 Matrix Set Representations and Operations

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. 9.2. CORA supports the following matrix set representations:

- Matrix polytope (Sec. 8.1)
- Matrix zonotope (Sec. 8.2); specialization of a matrix polytope.
- Interval matrix (Sec. 8.3); specialization of a matrix zonotope.

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. 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. Important operations for matrix sets are

- `display`: Displays the parameters of the set in the MATLAB workspace.
- `mtimes`: Overloads the '`*`' operator for the multiplication of various objects with a matrix set. For instance if M_set is a matrix set of proper dimension and Z is a zonotope, `M_set * Z` returns the linear map $\{Mx | M \in M\_set, x \in Z\}$.
- `plus`: Overloads the '`+`' operator for the addition of various objects with a matrix set. For instance if $M1\_set$ and $M2\_set$ are interval matrices of proper dimension, `M1_set + M2_set` returns the Minkowski sum $\{M1 + M2 | M1 \in M1\_set, M2 \in M2\_set\}$.
- `expm`: Returns the set of matrix exponentials for a matrix set.
- `intervalMatrix`: Computes an enclosing interval matrix.
- `vertices`: Returns the vertices of a matrix set.



(a) `plot(Z)`

(b) `plot(Z, [1,2], 'r:')`

(c) `plotFilled(Z, [1,2], 'w', 'EdgeColor', 'b')`

(d) `plotFilled(Z, [1,2], [.75 .75 .75], 'EdgeColor', 'none')`

(e) `plotFilled(Z, [1,2], [.6 .6 .6], 'EdgeColor', 'none')`

(f) `plot(Z, [1,2], 'k-', 'lineWidth', 3)`

Figure 17: Selection of different plot styles.

8.1 Matrix Polytopes

A matrix polytope is analogously defined as a V-polytope (see Sec. 7.6):

$$\mathcal{A}_{[p]} = \Big\{ \sum_{i=1}^{r} \alpha_i V^{(i)} \Big| \alpha_i \in \mathbb{R}, \alpha_i \geq 0, \sum_i \alpha_i = 1 \Big\}, \quad V^{(i)} \in \mathbb{R}^{n \times n}. \tag{9}$$

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. 7.6).

We support the following methods for polytope matrices:

- `display` – standard method, see Sec. 8.
- `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.
- `intervalMatrix` – standard method, see Sec. 8.
- `matPolytope` – constructor of the class.
- `matZonotope` – computes an enclosing matrix zonotope of a matrix polytope analogously to `zonotope` of the `vertices` class.

- `mpower` – overloaded '^' operator for the power of matrix polytopes.
- `mtimes` – standard method, see Sec. 8 for numeric matrix multiplication or multiplication with another matrix polytope.
- `plot` – plots 2-dimensional projection of a matrix polytope.
- `plus` – standard method, see Sec. 8. Addition is carried out for two matrix polytopes or a matrix polytope with a matrix.
- `polytope` – converts a matrix polytope to a polytope.
- `simplePlus` – computes the Minkowski addition of two matrix polytopes without reducing the vertices by a convex hull computation.
- `vertices` – standard method, see Sec. 8.

Since the matrix polytope class is written using the new structure for object oriented programming in MATLAB, it has the following public properties:

- `dim` – dimension.
- `verts` – number of vertices.
- `vertex` – cell array of vertices $V^{(i)}$ according to (9).

8.1.1 Matrix Polytope Example

The following MATLAB code demonstrates some of the introduced methods:

```
1  P1{1} = [1 2; 3 4]; % 1st vertex of matrix polytope P1
2  P1{2} = [2 2; 3 3]; % 2nd vertex of matrix polytope P1
3  matP1 = matPolytope(P1); % instantiate matrix polytope P1
4
5  P2{1} = [-1 2; 2 -1]; % 1st vertex of matrix polytope P2
6  P2{2} = [-1 1; 1 -1]; % 2nd vertex of matrix polytope P2
7  matP2 = matPolytope(P2); % instantiate matrix polytope P2
8
9  matP3 = matP1 + matP2 % perform Minkowski addition and display result
10 matP4 = matP1 * matP2 % compute multiplication of and display result
11
12 intP = intervalMatrix(matP1) % compute interval matrix and display result
```

This produces the workspace output

```
dimension:
     2

nr of vertices:
     4

vertices:
     0     4
     5     3

---------------
     0     3
     4     3
```

```
--------------
     1     4
     5     2

--------------
     1     3
     4     2

--------------
dimension:
     2

nr of vertices:
     4

vertices:
     3     0
     5     2

--------------
     1    -1
     1    -1

--------------
     2     2
     3     3

--------------
     0     0
     0     0

--------------
dimension:
     2

left limit:
     1     2
     3     3

right limit:
     2     2
     3     4
```

8.2 Matrix Zonotopes

A matrix zonotope is defined analogously to zonotopes (see Sec. 7.1):

$$\mathcal{A}_{[z]} = \left\{ G^{(0)} + \sum_{i=1}^{\kappa} p_i G^{(i)} \middle| p_i \in [-1, 1] \right\}, \quad G^{(i)} \in \mathbb{R}^{n \times n} \tag{10}$$

and is written in short form as $\mathcal{A}_{[z]} = (G^{(0)}, G^{(1)}, \dots, G^{(\kappa)})$, 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 $g^{(i)} \in \mathbb{R}^n$, one obtains a zonotope (see e.g., [17]).

We support the following methods for zonotope matrices:

- `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 [21, Proposition 4.3].
- `display` – standard method, see Sec. 8.
- `dominantVertices` – computes the dominant vertices of a matrix zonotope according to a heuristics.
- `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 [21, Section 4.4.4].
- `expmOneParam` – operator for the exponential matrix of a matrix zonotope when only one parameter is uncertain as described in [38, Theorem 1].
- `expmVertex` – computes the exponential matrix for a selected number of dominant vertices obtained by the `dominantVertices` method.
- `intervalMatrix` – standard method, see Sec. 8.
- `matPolytope` – converts a matrix zonotope into a matrix polytope representation.
- `matZonotope` – constructor of the class.
- `mpower` – overloaded '^' operator for the power of matrix zonotopes.
- `mtimes` – standard method, see Sec. 8 for numeric matrix multiplication or a multiplication with another matrix zonotope according to [21, Equation 4.10].
- `norm` – computes exactly the maximum norm value of all possible matrices.
- `plot` – plots 2-dimensional projection of a matrix zonotope.
- `plus` – standard method (see Sec. 8) for a matrix zonotope or a numerical matrix.
- `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.
- `uniformSampling` – creates samples uniformly within a matrix zonotope.
- `vertices` – standard method, see Sec. 8.

- `volume` – computes the volume of a matrix zonotope by computing the volume of the corresponding zonotope.
- `zonotope` – converts a matrix zonotope into a zonotope.

Since the matrix zonotope class is written using the new structure for object oriented programming in MATLAB, it has the following public properties:

- `dim` – dimension.
- `gens` – number of generators.
- `center` – $G^{(0)}$ according to (10).
- `generator` – cell array of matrices $G^{(i)}$ according to (10).

8.2.1 Matrix Zonotope Example

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

```
dimension:
     2

nr of generators:
     2

center:
     0     4
     5     3

generators:
     1     0
     1     1

---------------
         0    0.5000
    0.5000         0

---------------
dimension:
     1
```

```
nr of generators:
     3

center:
     3     0
     5     2

generators:
    1.0000    0.5000
    2.0000    1.5000

---------------
    -1     2
     1     1

---------------
         0    0.5000
    0.5000    0.5000

---------------
dimension:
     2

left limit:
     0     2
     2     3

right limit:
     2     2
     4     5
```

8.3 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 $\mathcal{A}_{[i]}$.

We support the following 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 [21, Proposition 4.4].
- `display` – standard method, see Sec. 8.
- `dominantVertices` – computes the dominant vertices of an interval matrix zonotope according to a heuristics.
- `exactSquare` – computes the exact square of an interval matrix.

- `expm` – operator for the exponential matrix of an interval matrix, evaluated dependently.
- `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 [21, Theorem 4.2].
- `expmVertex` – computes the exponential matrix for a selected number of dominant vertices obtained by the `dominantVertices` method.
- `exponentialRemainder` – returns the remainder of the exponential matrix according to [21, Proposition 4.1].
- `interval` – converts an interval matrix to an interval.
- `intervalMatrix` – constructor of the class.
- `matPolytope` – converts an interval matrix to a matrix polytope.
- `matZonotope` – converts an interval matrix to a matrix zonotope.
- `mpower` – overloaded '^' operator for the power of interval matrices.
- `mtimes` – standard method, see Sec. 8 for numeric matrix multiplication or a multiplication with another interval matrix according to [21, Equation 4.11].
- `norm` – computes exactly the maximum norm value of all possible matrices.
- `plot` – plots 2-dimensional projection of an interval matrix.
- `plus` – standard method, see Sec. 8. Addition is realized for two interval matrices or an interval matrix with a matrix.
- `powers` – computes the powers of an interval matrix up to a certain order.
- `randomSampling` – creates random samples within a matrix zonotope.
- `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.
- `subsref` – overloads the operator that selects elements, e.g., A(1,2), where the element of the first row and second column is referred to.
- `vertices` – standard method, see Sec. 8.
- `volume` – computes the volume of an interval matrix by computing the volume of the corresponding interval.

8.3.1 Interval Matrix Example

The following MATLAB code demonstrates some of the introduced methods:

```
Mcenter = [1 2; 3 4]; % center of interval matrix M1
Mdelta = [1 0; 1 1]; % delta of interval matrix M1
intM1 = intervalMatrix(Mcenter, Mdelta); % instantiate interval matrix M1

Mcenter = [-1 2; 2 -1]; % center of interval matrix M2
Mdelta = [0 0.5; 0.5 0]; % delta of interval matrix M2
intM2 = intervalMatrix(Mcenter, Mdelta); % instantiate interval matrix M2

intM3 = intM1 + intM2 % perform Minkowski addition and display result
intM4 = intM1 * intM2 % compute multiplication of and display result

matZ = matZonotope(intM1) % compute matrix zonotope and display result
```

This produces the workspace output

```
dimension:
     2

left limit:
   -1.0000    3.5000
    3.5000    2.0000

right limit:
    1.0000    4.5000
    6.5000    4.0000

dimension:
     2

left limit:
    1.0000   -3.0000
   -0.5000   -3.0000

right limit:
    5.0000    3.0000
   10.5000    7.0000

dimension:
     2

nr of generators:
     3

center:
     1     2
     3     4

generators:
     1     0
     0     0

---------------
     0     0
```

```
     1     0

---------------
     0     0
     0     1

---------------
```

9 Continuous Dynamics

This section introduces various classes to compute reachable sets of continuous and hybrid dynamics. One can directly compute reachable sets for each class, or include those classes into a hybrid automaton for the reachability analysis of hybrid systems. Note that besides reachability analysis, the simulation of particular trajectories is also supported. CORA supports the following continuous dynamics:

- Linear systems (Sec. 9.1)
- Linear systems with uncertain, fixed or varying parameters (Sec. 9.2)
- Linear probabilistic systems (Sec. 9.3)
- Nonlinear systems (Sec. 9.4)
- Nonlinear systems with uncertain fixed parameters (Sec. 9.6)
- Nonlinear differential-algebraic systems (Sec. 9.7)

Each class for continuous dynamics inherits from the parent class `contDynamics`. This class itself is inherited from the `matlab.mixin.Copyable` class, which is an abstract handle class that provides a copy method for copying handle objects. This implied that objects created from this class only reference the object data instead of reserving dedicated memory (call by reference). Copying an object creates another reference to the same data. To create a true copy, the `copy` method has to be used. Since for reachability analysis, multiple instances of the same dynamics are not required, the instantiation from a `matlab.mixin.Copyable` class makes sense since one does not have to pass the changed object for each called method. The continuous set classes, which inherit from `contSet` are not using this concept, since sets have to be copied even for simple operations, such as $Z_3 = Z_1 + Z_2$, where Z_i are zonotope objects.

The parent class provides the following methods:

- `derivatives` – computes multivariate derivatives (jacobians, hessians, etc.) of nonlinear systems in a symbolic way.
- `dimension` – returns the dimension of the system.
- `display` – displays the parameters of the parent class in the MATLAB workspace.
- `reach` – computes the reachable set for the entire time horizon.
- `simulate_random` – performs several random simulations of the system. It can be set how many simulations should be performed, what percentage of initial states should start at vertices of the initial set, what percentage of inputs should be chosen from vertices of the input set, and how often the input should be changed.
- `simulate_rrt` – simulates a system using rapidly exploring random trees.
- `symVariables` – generates symbolic variables of a continuous system.

In addition, each class realizes these methods:

- `display`: Displays the parameters of particular continuous dynamics beyond the information of the parent class in the MATLAB workspace.
- `initReach`: Initializes the reachable set computation.
- `post`: Computes the reachable set for the next time interval.
- `simulate`: Produces a single trajectory that numerically solves the system for a particular initial state and a particular input trajectory.

There exist some further auxiliary methods for each class, but those are not relevant unless one aims to change details of the provided algorithms. In contrast to the set representations, all continuous systems have the same methods, therefore the methods are not listed for the individual classes. We mainly focus on the method `initReach`, which is computed differently for each class.

9.1 Linear Systems

The most basic system dynamics considered in this software package are linear systems of the form

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) + c \\ y(t) &= Cx(t) + Du(t) + k, \end{aligned} \tag{11}$$

where $x(0) \in \mathcal{X}_0 \subset \mathbb{R}^n$, $u(t) \in \mathcal{U} \subset \mathbb{R}^m$, $A \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{n \times m}$, $c \in \mathbb{R}^n$, $C \in \mathbb{R}^{p \times n}$, $D \in \mathbb{R}^{p \times m}$ and $k \in \mathbb{R}^p$. For the computation of reachable sets, we use the equivalent system

$$\dot{x}(t) = Ax(t) + \tilde{u}(t) + c, \quad x(0) \in \mathcal{X}_0 \subset \mathbb{R}^n, \quad \tilde{u}(t) \in \tilde{\mathcal{U}} = B \otimes \mathcal{U} \subset \mathbb{R}^n, \quad c \in \mathbb{R}^n \tag{12}$$

where $\mathcal{E} \otimes \mathcal{F} = \{E\,F | E \in \mathcal{E}, F \in \mathcal{F}\}$ is the set-based multiplication (one argument can be a singleton).

9.1.1 Method `initReach`

The method `initReach` computes the required steps to obtain the reachable set for the first point in time r and the first time interval $[0, r]$ as follows. Given is the linear system in (12). For further computations, we introduce the center of the set of inputs u_c and the deviation from the center of $\tilde{\mathcal{U}}$, $\tilde{\mathcal{U}}_\Delta := \tilde{\mathcal{U}} \oplus (-u_c)$. According to [18, Section 3.2], the reachable set for the first time interval $\tau_0 = [0, r]$ is computed as shown in Fig. 18:

1. Starting from $\mathcal{X}_0$, compute the set of all solutions $\mathcal{R}_h^d$ for the affine dynamics $\dot{x}(t) = Ax(t) + u_c$ at time r.
2. Obtain the convex hull of $\mathcal{X}_0$ and $\mathcal{R}_h^d$ to approximate the reachable set for the first time interval τ_0.
3. Compute $\mathcal{R}^d(\tau_0)$ by enlarging the convex hull, firstly to bound all affine solutions within τ_0 and secondly to account for the set of uncertain inputs $\tilde{\mathcal{U}}_\Delta$.

The following private functions take care of the required computations:

- `exponential` – computes an over-approximation of the matrix exponential e^{Ar} based on the Lagrangian remainder as proposed in [39, Proposition 2]. A more conservative approach previously used [18, Equation 3.2,3.3].

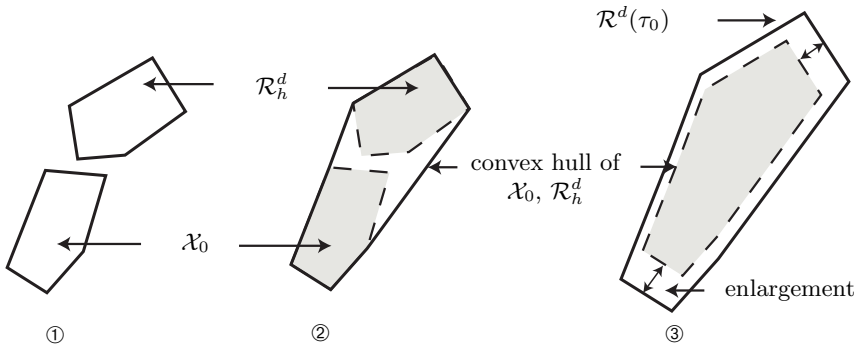


Figure 18: Steps for the computation of an over-approximation of the reachable set for a linear system.

- `tie` (**t**ime **i**nterval **e**rror) – computes the error made by generating the convex hull of reachable sets of points in time for the reachable set of the corresponding time interval as described in [39, Section 4]. A more conservative approach previously used [18, Proposition 3.1], which can only be used in combination with [18, Equation 3.2,3.3].
- `inputSolution` – computes the reachable set due to the input according to the superposition principle of linear systems. The computation is performed as suggested in [18, Theorem 3.1]. As noted in [39, Theorem 2], it is required that the input set is convex. The error term in [39, Theorem 2] is slightly better, but is computationally more expensive so that the algorithm form [18, Theorem 3.1] is used.

9.1.2 Method `post`

We have implemented two different methods for propagating the reachable sets for each time interval in the method `post`. The wrapping-free approach in [40] (set `options.linAlg = 1`) and the approach in [17] (set `options.linAlg = 2` or do not specify `options.linAlg`; default setting). The wrapping-free approach is computationally more efficient and has no wrapping effect. However, since partial input solution sets are over-approximated by intervals, for some examples, one obtains overly conservative results. For this reason, we have set the method in [17] as the default method (the method in [40] was the default method in the 2016 release).

9.2 Linear Systems with Uncertain, Fixed or Varying 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.

Fixed parameters A linear system with uncertain parameters that are fixed over time can be written as

$$\dot{x}(t) = A(p)x(t) + \tilde{u}(t), \quad x(0) \in \mathcal{X}_0 \subset \mathbb{R}^n, \quad p \in \mathcal{P}, \quad \tilde{u}(t) \in \tilde{\mathcal{U}} = \{B(p) \otimes \mathcal{U} | \mathcal{U} \subset \mathbb{R}^n, p \in \mathcal{P}\}, \tag{13}$$

The approach for fixed parameter values is presented in [18]. To execute this algorithm, one has to set `paramType='constParam'` or not set this parameter since it is the default setting (example: `linSys = linParamSys(A, B, r, maxOrder)`).

Varying parameters When the parameters are time-varying, we obtain

$$\dot{x}(t) = A(t)x(t) + \tilde{u}(t), \quad x(0) \in \mathcal{X}_O \subset \mathbb{R}^n, \quad A(t) \in \mathcal{A}, \quad \tilde{u}(t) \in \tilde{\mathcal{U}}.$$

The approach for fixed parameter values is presented in [39]. To execute this algorithm, one has to set `paramType='varParam'` (example: `linSys = linParamSys(A, B, r, maxOrder, 'varParam')`).

Alternative computation 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. 9.4. The problem 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.

For further explanations, we introduce the set of state and input matrices as

$$\mathcal{A} = \{A(p)|p \in \mathcal{P}\}, \quad \mathcal{B} = \{B(p)|p \in \mathcal{P}\}. \tag{14}$$

The set of state matrices can be represented by any matrix set introduced in Sec. 8.

Since the `linParamSys` class is written using the new structure for object oriented programming in MATLAB, it has the following public properties:

- `A` – set of system matrices $\mathcal{A}$, see (14). The set of matrices can be represented by any matrix set introduced in Sec. 8.
- `B` – set of input matrices $\mathcal{B}$, see (14). The set of matrices can be represented by any matrix set introduced in Sec. 8.
- `stepSize` – constant step size $t_{k-1} - t_k$ for time intervals of the reachable set computation.
- `taylorTerms` – number of Taylor terms for computing the matrix exponential, see [18, Theorem 3.2].
- `mappingMatrixSet` – set of exponential matrices, see Sec. 8.
- `E` – remainder of matrix exponential computation.
- `F` – uncertain matrix to bound the error for time interval solutions, see e.g., [18, Proposition 3.1].
- `inputF` – uncertain matrix to bound the error for time interval solutions of inputs, see e.g., [18, Proposition 3.4].
- `inputCorr` – additional uncertainty of the input solution if origin is not contained in input set, see [18, Equation 3.9].
- `Rinput` – reachable set of the input solution, see Sec. 9.1.
- `Rtrans` – reachable set of the input u_c, see Sec. 9.1.
- `RV` – reachable set of the input $\tilde{\mathcal{U}}_\Delta$, see Sec. 9.1.
- `sampleMatrix` – possible matrix $\hat{A}$ such that $\hat{A} \in \mathcal{A}$.

9.2.1 Method `initReach`

The method `initReach` computes the reachable set for the first point in time r and the first time interval $[0, r]$ similarly as for 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 $\mathcal{B}$. The initial state solution becomes

$$\mathcal{R}_h^d = e^{\mathcal{A}r}\mathcal{X}_0 = \{e^{Ar}x_0 | A \in \mathcal{A}, x_0 \in \mathcal{X}_0\}. \tag{15}$$

Similarly, the reachable set due to the input solution changes as described in [18, Section 3.3]. The following private functions take care of the required computations:

- `mappingMatrix` – computes the set of matrices which map the states for the next point in time according to [21, Section 3.1].
- `tie` (**t**ime **i**nterval **e**rror) – computes the error made by generating the convex hull of reachable sets of points in time for the reachable set of the corresponding time interval as described in [21, Section 3.2].
- `inputSolution` – computes the reachable set due to the input according to the superposition principle of linear systems. The computation is performed as suggested in [21, Theorem 1].

9.3 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 [41]:

$$\begin{aligned} \dot{x} &= Ax(t) + u(t) + C\xi(t), \\ x(0) \in \mathbb{R}^n,\, u(t) &\in \mathcal{U} \subset \mathbb{R}^n,\, \xi \in \mathbb{R}^m, \end{aligned} \tag{16}$$

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. In the probabilistic setting ($C \neq 0$), the probability density function (PDF) at time $t = r$ of the random process $\mathbf{X}(t)$ defined by (16) for a specific trajectory $u(t) \in \mathcal{U}$ is denoted 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 (16) $\forall t \in [0, r]$, $u(t) \in \mathcal{U}$, $f_{\mathbf{X}}(x, 0) = f_0\}$. The enclosing probabilistic hull for a time interval is defined as $\bar{f}_{\mathbf{X}}(x, [0, r]) = \sup\{\bar{f}_{\mathbf{X}}(x, t) | t \in [0, r]\}$.

9.3.1 Method `initReach`

The method `initReach` computes the probabilistic reachable set for a first point in time r and the first time interval $[0, r]$ similarly to Sec. 9.1.1. The main difference is that we compute enclosing probabilistic hulls as defined above. The following private functions take care of the required computations:

- `pexpm` – computes the over-approximation of the exponential of a system matrix similarly as for linear systems in Sec. 9.1.
- `tie` (**t**ime **i**nterval **e**rror – computes the `tie` similarly as for linear systems in Sec. 9.1.
- `inputSolution` – computes the reachable set due to the input according to the superposition principle of linear systems. The computation is performed as suggested in [26, Sec. VI.B].

9.4 Nonlinear Systems

So far, reachable sets of linear continuous systems have been presented. Although a fairly large group of dynamic systems can be described by linear continuous systems, the extension to nonlinear continuous systems is an important step for the analysis of more complex systems. The analysis of nonlinear systems is much more complicated since many valuable properties are no longer valid. One of them is the superposition principle, which allows the homogeneous and the inhomogeneous solution to be obtained separately. Another is that reachable sets of linear systems can be computed by a linear map. 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 abstraction. We present abstraction by linear systems as presented in [18, Section 3.4] and by polynomial systems as presented in [24]. Since the abstraction causes additional errors, the abstraction errors are determined in an over-approximative way and added as an additional uncertain input so that an over-approximative computation is ensured.

General nonlinear continuous systems with uncertain parameters and Lipschitz continuity are considered. In analogy to the previous linear systems, the initial state $x(0)$ can take values from a set $\mathcal{X}_0 \subset \mathbb{R}^n$ and the input u takes values from a set $\mathcal{U} \subset \mathbb{R}^m$. The evolution of the state x is defined by the following differential equation:

$$\dot{x}(t) = f(x(t), u(t)), \qquad x(0) \in \mathcal{X}_0 \subset \mathbb{R}^n, \quad u(t) \in \mathcal{U} \subset \mathbb{R}^m,$$

where $u(t)$ and $f(x(t), u(t))$ are assumed to be globally Lipschitz continuous so that the Taylor expansion for the state and the input can always be computed, a condition required for the abstraction.

A brief visualization of the overall concept for computing the reachable set is shown in Fig. 19. 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:

① The nonlinear system $\dot{x}(t) = f(x(t), u(t))$ is either abstracted to a linear system as shown in (12), or after introducing $z = [x^T,\, u^T]^T$, to a polynomial system of the form

$$\begin{aligned} \dot{x}_i = f^{abstract}(x,u) = & w_i + \frac{1}{1!}\sum_{j=1}^{o} C_{ij} z_j(t) + \frac{1}{2!}\sum_{j=1}^{o}\sum_{k=1}^{o} D_{ijk} z_j(t) z_k(t) \\ & + \frac{1}{3!}\sum_{j=1}^{o}\sum_{k=1}^{o}\sum_{l=1}^{o} E_{ijkl} z_j(t) z_k(t) z_l(t) + \ldots \end{aligned} \tag{17}$$

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.

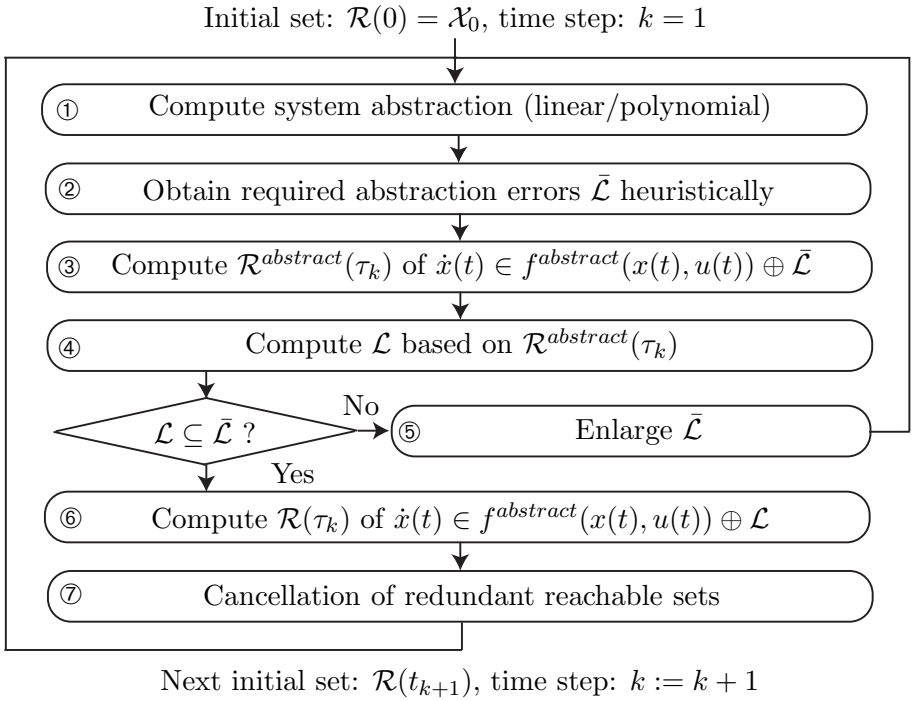


Figure 19: Computation of reachable sets – overview.

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 When $\mathcal{L} \not\subseteq \bar{\mathcal{L}}$, 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 from then on.

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`. The ratio of the current abstraction errors to the allowed abstraction errors is indicated in the workspace outputs using the keyword `perfInd`. If `perfInd >= 1`, the reachable set has to be split.

9.4.1 Method `initReach`

The method `initReach` computes the reachable set for a first point in time r and the first time interval $[0, r]$. In contrast to linear systems, it is required to call `initReach` for each time interval τ_k since the system is abstracted for each time interval τ_k by a different abstraction $f^{abstract}(x, u)$.

The following functions take care of the most relevant computations:

- `linReach` – computes the reachable set of the abstraction $f^{abstract}(x(t), u(t)) \oplus \bar{\mathcal{L}}$ and returns if the initial set has to be split in order to control the abstraction error. The name

of the function has historical reasons and will change.

- `linearize` – linearizes the nonlinear system.
- `linError_mixed_noInt` – computes the linearization error without use of interval arithmetic according to [19, Theorem 1].
- `linError_thirdOrder` – computes abstraction errors according to [24, Section 4.1].
- `linError` – easiest, but also most conservative computation of the linearization error according to [9, Proposition 1].

9.5 Discrete-time Nonlinear Systems

Unlike all other systems, the class `nonlinearSysDT` considers discrete time models. In more detail, the class represents nonlinear discrete time systems, which are defined by the following equation:

$$x(k+1) = f\left(x(k), u(k)\right), \quad x(0) \in \mathcal{X}_0 \subset \mathbb{R}^n, \;\; \forall k : u(k) \in \mathcal{U} \subset \mathbb{R}^m$$

The class provides functionality for simulation as well as for reachable set computation.

9.5.1 Method `reach`

This method computes the reachable set for the specified time horizon. 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 $f(\cdot)$, where the sets $\mathcal{X}_k$ and $\mathcal{U}$ are the function inputs. Similar to continuous-time nonlinear systems, CORA applies the conservative linearization technique to calculate the image of the function. More specifically, we first generate the Taylor series of the nonlinear function $f\left(x(k), u(k)\right)$ up to a certain order p, using the center of the sets $\mathcal{X}_k$ and $\mathcal{U}$ as the combined expansion point. For all summands of the Taylor series we compute the set of possible values for $x(k) \in \mathcal{X}_k$ and $u(k) \in \mathcal{U}_k$. The set-valued summands are then added using Minkowski addition, which results in an approximation of the image of the function. Finally, in order to guarantee that the calculated reachable set is an over-approximation, we have to consider the Lagrange remainder of order $p+1$, which we evaluate with interval arithmetics. There exist some important settings that significantly influence the tightness of the calculated reachable set:

- `options.taylorOrder`: Number of considered Taylor series terms p. An increase in the number of terms usually leads to a tighter reachable set. The maximum value that is supported by CORA is `options.taylorOrder = 3`.
- `options.errorOrder`: Zonotope order to which the set $\mathcal{X}_k$ is reduced before the quadratic or higher order terms of the Taylor series are evaluated. The reduction is necessary since the evaluation of quadratic or higher order terms for a zonotope results in a large increase in the number of zonotope generators.

Computing the Minkowski sum as well as the evaluation of quadratic or higher order Taylor series terms leads to zonotopes with a quickly increasing number of generators. In order to keep the runtime of the reachability algorithm reasonable, we therefore reduce the zonotope order of the set $\mathcal{X}_k$ after each time step to the user-defined value specified in `options.zonotopeOrder`.

Note: We have not implemented a dedicated class for computing reachable sets of linear discrete time systems. Given the linear discrete-time system $x(k+1) =$

$Ax(k) + u(k)$, $\forall k : u(k) \in \mathcal{U}$, one can simply obtain the reachable sets using the following code: `R{i + 1} = A * R{i} + U`. A full example can be found under `CORA/examples/contDynamics/linear/example_linear_reach_05_discreteTime.m`.

9.6 Nonlinear Systems with Uncertain Fixed Parameters

The class `nonlinParamSys` extends the class `nonlinearSys` by considering uncertain parameters p:

$$\dot{x}(t) = f(x(t), u(t), p), \qquad x(0) \in \mathcal{X}_O \subset \mathbb{R}^n, \quad u(t) \in \mathcal{U} \subset \mathbb{R}^m, \quad p \in \mathcal{P} \subset \mathbb{R}^p.$$

The functionality provided is identical to `nonlinearSys`, except that the abstraction to polynomial systems is not yet implemented.

9.7 Nonlinear Differential-Algebraic Systems

The class `nonlinDASys` considers time-invariant, semi-explicit, index-1 DAEs. Parametric uncertainties as demonstrated in Sec. 9.6 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. After introducing the vectors of differential variables x, algebraic variables y, and inputs u, the semi-explicit DAE can generally be written as

$$\begin{aligned} \dot{x} &= f(x(t), y(t), u(t)) \\ 0 &= g(x(t), y(t), u(t)), \\ [x^T(0), y^T(0)]^T &\in \mathcal{R}(0), \quad u(t) \in \mathcal{U}, \end{aligned} \tag{18}$$

where $\mathcal{R}(0)$ over-approximates the set of consistent initial states and $\mathcal{U}$ is the set of possible inputs. 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 [42, 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 [43, 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 [42, Chapter 9.1].

To apply the methods presented in the previous section to compute reachable sets for DAEs, an abstraction of the original nonlinear DAEs to linear differential inclusions is performed for each consecutive time interval τ_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 [19, 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 [19, Proposition 2].

The `nonlinDASys` class has the following public properties:

- `dim` – number of dimensions.
- `nrOfConstraints` – number of constraints.
- `nrOfInputs` – number of inputs.
- `dynFile` – handle to the m-file containing the dynamic function $f(x(t), y(t), u(t))$.

- `conFile` – handle to the m-file containing the constraint function $g(x(t), y(t), u(t))$.
- `jacobian` – handle to the m-file containing the Jacobians of the dynamic function and the constraint function.
- `hessian` – handle to the m-file containing the Hessians of the dynamic function and the constraint function.
- `thirdOrderTensor` – handle to the m-file containing the third order tensors of the dynamic function and the constraint function.
- `linError` – handle to the m-file containing the Lagrangian remainder.
- `other` – other information.

10 Hybrid Dynamics

In CORA, hybrid systems are modeled by hybrid automata. Besides a continuous state x, there also exists a discrete state v for hybrid systems. The continuous initial state may take values within continuous sets while only a single initial discrete state is assumed without loss of generality[7]. 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 Sec. 15.2.1), where the velocity of the ball changes instantaneously when hitting the ground.

The formal definition of the hybrid automaton is similarly defined as in [37]. The main difference is the consideration of uncertain parameters and the restrictions on jumps and guard sets. A hybrid automaton $HA = (\mathcal{V}, v^0, \mathcal{X}, \mathcal{X}^0, \mathcal{U}, \mathcal{P}, \texttt{inv}, \texttt{T}, \texttt{g}, \texttt{h}, \texttt{f})$, as it is considered in CORA, consists of:

- the finite set of locations $\mathcal{V} = \{v_1, \ldots, v_\xi\}$ with an initial location $v^0 \in \mathcal{V}$.
- the continuous state space $\mathcal{X} \subseteq \mathbb{R}^n$ and the set of initial continuous states $\mathcal{X}^0$ such that $\mathcal{X}^0 \subseteq \texttt{inv}(v^0)$.
- the continuous input space $\mathcal{U} \subseteq \mathbb{R}^m$.
- the parameter space $\mathcal{P} \subseteq \mathbb{R}^p$.
- the mapping[8] $\texttt{inv}$: $\mathcal{V} \to 2^{\mathcal{X}}$, which assigns an invariant $\texttt{inv}(v) \subseteq \mathcal{X}$ to each location v.
- the set of discrete transitions $\texttt{T} \subseteq \mathcal{V} \times \mathcal{V}$. A transition from $v_i \in \mathcal{V}$ to $v_j \in \mathcal{V}$ is denoted by (v_i, v_j).
- the guard function $\texttt{g} : \texttt{T} \to 2^{\mathcal{X}}$, which associates a guard set $\texttt{g}((v_i, v_j))$ for each transition from v_i to v_j, where $\texttt{g}((v_i, v_j)) \cap \texttt{inv}(v_i) \neq \emptyset$.
- the jump function $\texttt{h} : \texttt{T} \times \mathcal{X} \to \mathcal{X}$, which returns the next continuous state when a transition is taken.
- the flow function $\texttt{f} : \mathcal{V} \times \mathcal{X} \times \mathcal{U} \times \mathcal{P} \to \mathbb{R}^n$, which defines a continuous vector field for the time derivative of x: $\dot{x} = \texttt{f}(v, x, u, p)$.

[7] In the case of several initial discrete states, the reachability analysis can be performed for each discrete state separately.

[8] $2^{\mathcal{X}}$ is the power set of $\mathcal{X}$.

The invariants $\texttt{inv}(v)$ and the guard sets $\texttt{g}((v_i, v_j))$ are modeled by polytopes. The jump function is restricted to a linear map

$$x' = K_{(v_i,v_j)}\, x + l_{(v_i,v_j)}, \tag{19}$$

where x' denotes the state after the transition is taken and $K_{(v_i,v_j)} \in \mathbb{R}^{n\times n}$, $l_{(v_i,v_j)} \in \mathbb{R}^n$ are specific for a transition (v_i, v_j). The input sets $\mathcal{U}_v$ are modeled by zonotopes and are also dependent on the location v. Note that in order to use the results from reachability analysis of nonlinear systems, the input $u(t)$ is assumed to be locally Lipschitz continuous. The set of parameters $\mathcal{P}_v$ can also be chosen differently for each location v.

The evolution of the hybrid automaton is described informally as follows. Starting from an initial location $v(0) = v^0$ and an initial state $x(0) \in \mathcal{X}^0$, the continuous state evolves according to the flow function that is assigned to each location v. If the continuous state is within a guard set, the corresponding transition can be taken and has to be taken if the state would otherwise leave the invariant $\texttt{inv}(v)$. When the transition from the previous location v_i to the next location v_j is taken, the system state is updated according to the jump function and the continuous evolution within the next invariant.

Because the reachability of discrete states is simply a question of determining if the continuous reachable set hits certain guard sets, the focus of CORA is on the continuous reachable sets. Clearly, as for the continuous systems, the reachable set of the hybrid system has to be over-approximated in order to verify the safety of the system. An illustration of a reachable set of a hybrid automaton is given in Fig. 20.

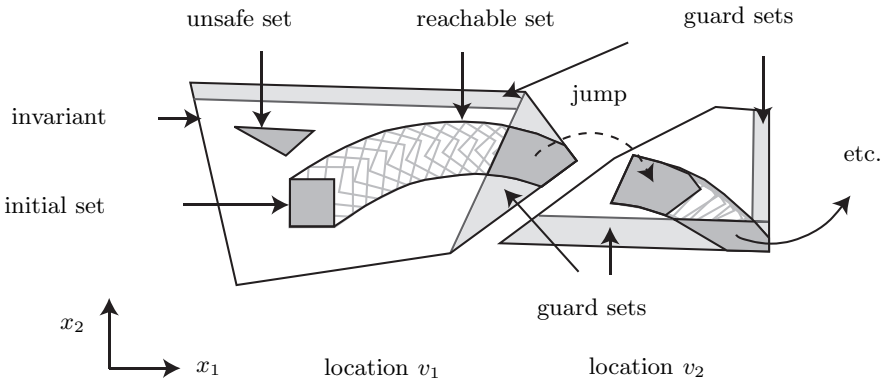


Figure 20: Illustration of the reachable set of a hybrid automaton.

10.1 Simulation of Hybrid Automata

While the reachable set computation of hybrid systems as performed in CORA is described in several publications, see e.g., [18,44,45], 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. 12(b). This is performed by transforming intervals, zonotopes, and zonotope bundles to a polytope, see Tab. 1. 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. 21.

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. 21(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. 22.

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.

10.2 Hybrid Automaton

A hybrid automaton is implemented as a collection of `locations`. We mainly support the following methods for hybrid automata:

- `hybridAutomaton` – constructor of the class.
- `plot` – plots the reachable set of the hybrid automaton.
- `reach` – computes the reachable set of the hybrid automaton.
- `simulate` – computes a hybrid trajectory of the hybrid automaton.

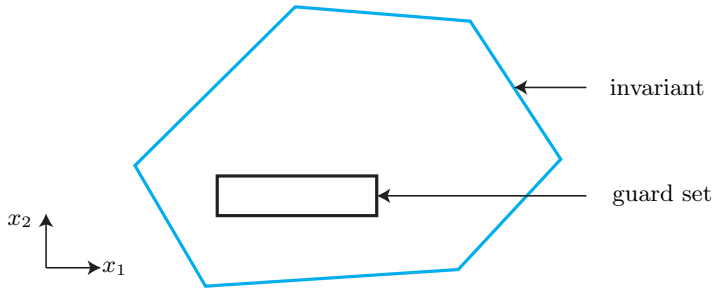
10.3 Location

Each location consists of:

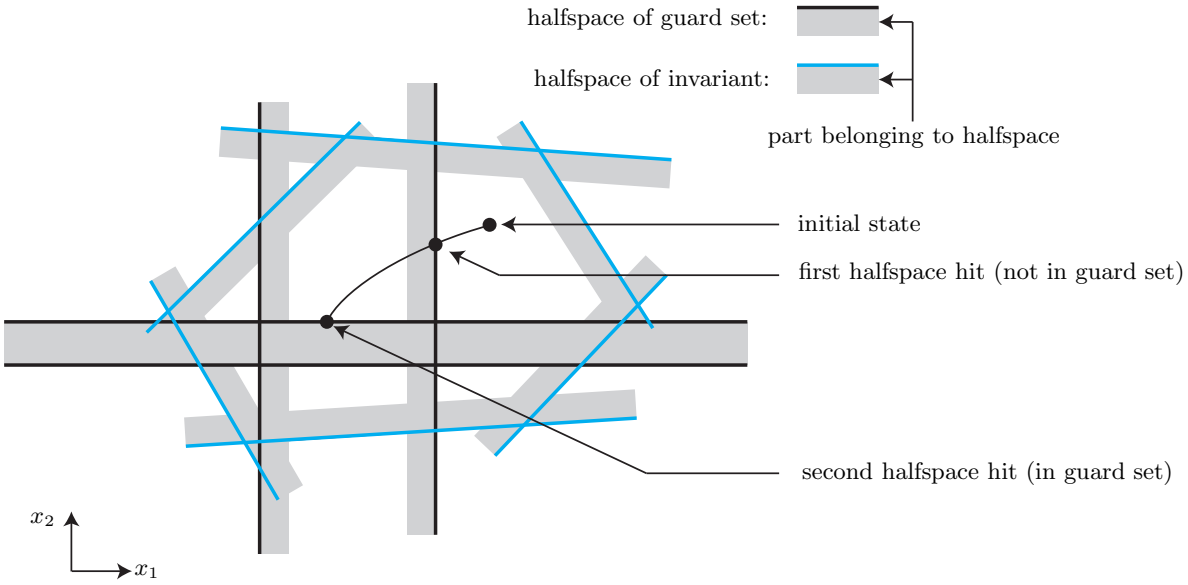
- `contDynamics` – specified by a continuous dynamics of Sec. 9.
- `id` – unique number of the location.
- `invariant` – specified by a set representation of Sec. 7.
- `name` – saved as a string describing the location.
- `transitions` – cell array of objects of the class `transition`.

We mainly support the following methods for locations:

- `display` – displays the parameters of the location in the MATLAB workspace.



(a) Considered location.



(b) Simulation using halfspaces.

Figure 21: Illustration of the algorithm for simulating a hybrid automaton.

- `enclosePolytopes` – encloses a set of polytopes using different over-approximating zonotopes.
- `guardIntersect` – intersects the reachable sets with potential guard sets and returns enclosing zonotopes for each guard set.
- `location` – constructor of the class.
- `potInt` – determines which reachable sets potentially intersect with guard sets of a location.
- `reach` – computes the reachable set for the location.
- `simulate` – produces a single trajectory that numerically solves the system within the location starting from a point rather than from a set.

10.4 Transition

Each transition consists of

- `guard` – specified by a set representation of Sec. 7.
- `inputLabel` – input event to communicate over events.

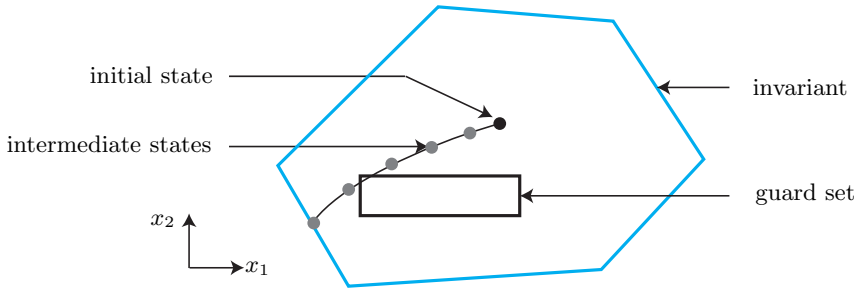


Figure 22: Guard intersections can be missed when one only checks whether intermediate states are in any guard set.

- `outputLabel` – output event to communicate over events.
- `reset` – struct containing the information for a linear reset.
- `target` – `id` of the target location when the transition occurs.

We mainly support the following methods for transitions:

- `display` – displays the parameters of the transition in the MATLAB workspace.
- `reset` – computes the reset map after a transition occurs (also called 'jump function').

10.5 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 for the overall system from the hybrid automaton models of the subcomponents (parallel composition, see e.g., [46, 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 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.

Parallel hybrid automata are implemented in CORA by the class `parallelHybridAutomaton`. For better illustration of the required information, we introduce the example presented in Fig. 23 consisting of three components. 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 [47]. 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. 14. To manually construct a `parallelHybridAutomaton` object, three arguments are required:

- `components` – list containing all subcomponents of the system. Each subcomponent has to be represented as a `hybridAutomaton` object

- `stateBinds` – list containing vectors that define the relation between the states of the subcomponents and the states of the composed system. In the example shown in Fig. 23, the full system has ten states, where the first component describes the evolution of the first three states x_1, x_2, x_3, the second component contains the states x_4, x_5, x_6, x_7, x_8, and the third one the states x_9, x_{10}. In CORA, this can be specified as follows:

```
stateBinds{1} = [1; 2; 3]; % states of component 1
stateBinds{2} = [4; 5; 6; 7; 8]; % states of component 2
stateBinds{3} = [9; 10]; % states of component 3
```

- `inputBinds` – list 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. In the example shown in Fig. 23, 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 input of component 2.

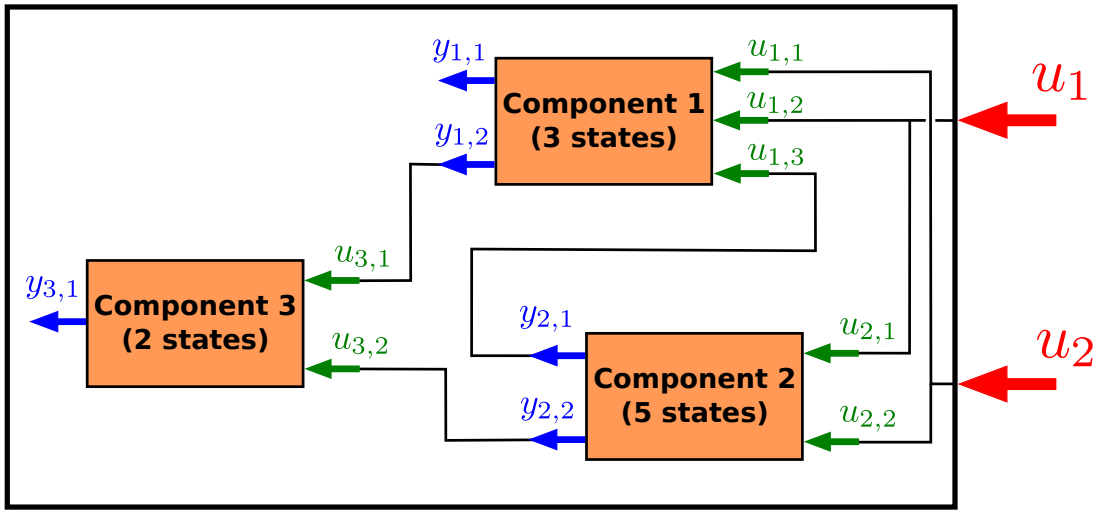


Figure 23: Example of a parallel hybrid automaton that consists of three subcomponents.

In addition to the general options for reachability analysis that are listed in Section 12, the `parallelHybridAutomaton` class requires some additional settings. These additional options are necessary to specify the system input for the composed system. CORA supports global and location dependent inputs. In the latter case, the subcomponent on whose location the input depends on has to be specified. All required additional settings for parallel hybrid automata are listed in the options struct as shown below:

- `options.inputCompMap` – vector specifying which system inputs are global (value 0) and which are location dependent (index of corresponding subcomponent). The value [0 3 0] for example indicates that the system inputs one and three are global, whereas the system

input two is different for each location of subcomponent 3.

- `options.uCompLoc` – cell-array containing the vector of system inputs for all locations of each subcomponent. Only necessary for simulation.
- `options.UCompLoc` – cell-array containing the set of uncertain inputs for all locations of each subcomponent. Necessary for reachability analysis and random simulation.
- `options.uCompLocTrans` – cell-array containing the vector with the offset of the input set for all locations of each subcomponent. Necessary for reachability analysis and random simulation.
- `options.uGlob` – vector specifying the global system inputs. Only necessary for simulation.
- `options.UGlob` – set of uncertain global system inputs. Necessary for reachability analysis and random simulation.
- `options.uGlobTrans` – vector specifying the offset for the uncertain global system inputs. Necessary for reachability analysis and random simulation.

11 Abstraction to Discrete Systems

11.1 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 polytope 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.
- `exactIntersectingSegments` – finds the exact segments of the partition that intersect a set P, and the proportion of P that is in each segment.
- `intersectingCells` – returns the cells possibly intersecting with a continuous set, over-approximatively, by overapproximating the convex set as a multidimensional interval.
- `nrOfCells` – returns the number of cells of the partition.
- `findSegments` – returns segment indices intersecting with a given interval hull.
- `nrOfStates` – returns the number of discrete states of the partition.
- `partition` – constructor of the class.

- `plot` – plots the partition.

The methods for the partition class have changed from the 2016 to the 2018 version. A table showing the difference can be found in Appendix A.

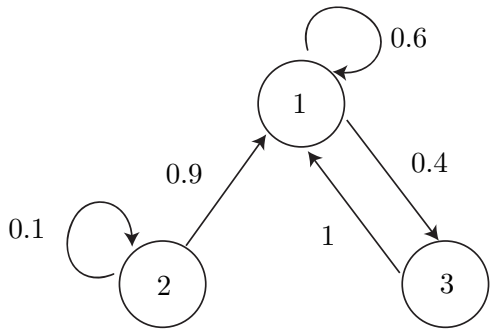
11.2 Abstraction to Markov Chains

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. 11.1. 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 [48]: 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 Ω 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. 24 by a graph whose nodes represent the states $1, 2, 3$ and whose labeled arrows represent the transition probabilities Φ_{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 24: 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 Φ. Thus, the continuous time at time step k is $t_k = k \cdot \tau$. The generation of a Markov chain from a continuous dynamics is performed as described in [18, 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 such that it can be used for an optimized update as presented in [49].
- `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.

11.3 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. [49, 50]. The probabilistic information allows 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. 11.2. 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 [18, 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 [18, Sec. 5], perform the following steps:

1. Run `/discrDynamics/ProbOccupancyPrediction/intersection/start_intersectionDatabase` to obtain an intersection database. The result is a structure `fArray`. 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. 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 driving braking 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. 25. For visualization reasons, the position distributions are plotted in separate plots, although the vehicles drive in the same lane. Dark regions indicate high probability, while bright regions represent areas of low probability. In order to improve the visualization, the colors are separately normalized for each vehicle.

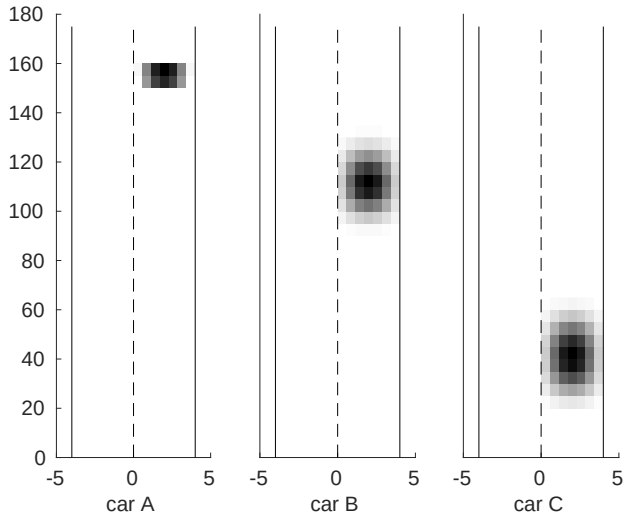


Figure 25: Probabilistic occupancy prediction of the braking scenario.

12 Options for Reachability Analysis

Most parameters for the computation of reachable sets are controlled by a `struct` called `options`. Please note that in most cases only a fraction of these options have to be specified. A good guidance on which options are required is provided by the examples in the folder `/examples`. In the next release we will provide user guidance on which options have to be set for which problem. The most important fields of the options struct are:

General

- `R0` – initial set of states.
- `reductionTechnique` –sets the reduction technique. The list of techniques can be found in `/contSet/zonotope/reduce.m`.

- `taylorTerms` – considered Taylor terms for the exponential matrix.
- `saveOrder` – maximum order of the zonotopes that are returned as output arguments (optional).
- `tFinal` – final time of the analysis.
- `timeStep` – step size $t_{k+1} - t_k$.
- `tStart` – start time of the analysis.
- `u` – constant input for simulations.
- `U` – uncertain input set $\tilde{\mathcal{U}}_\Delta$.
- `uTrans`/`uTransVec` – `uTrans`: translation of the uncertain input set $\tilde{\mathcal{U}}_\Delta$; `uTransVec`: varying u_c for each time step: translation of the uncertain input set $\tilde{\mathcal{U}}_\Delta$.
- `x0` – initial state.
- `zonotopeOrder` – maximum order of zonotopes.

Linear systems

- `linAlg` – flag to choose the wrapping-free approach in [40] (set `options.linAlg = 1`) or the approach in [17] (set `options.linAlg = 2` or do not specify `options.linAlg`; default setting) for reachability analysis of linear systems.
- `originContained` – flag whether the origin is contained in the set of uncertain inputs $\tilde{\mathcal{U}}$ (1: yes, 0: no).
- `outputOrder` – maximum order of the zonotopes that represent the set of system ouputs y (optional).

Nonlinear System

- `advancedLinErrorComp` – flag to enable advanced linearization error computation (1: on, 0: off).
- `errorOrder` – maximum zonotope order for the computation of nonlinear maps.
- `intermediateOrder` – order up to which no interval methods are used in matrix set computations.
- `maxError` – maximum allowed abstraction errors before a reachable set is split.
- `reductionInterval` – number of time steps after which redundant reachable sets are removed.
- `simplify` – method used to simplify the algebraic expressions in the Lagrange remainder term ('none', 'simplify' or 'collect', optional).
- `tensorOrder` – maximum order up to which tensors are considered in the abstraction of the system.
- `tensorParallel` – flag to enable evaluation of the Lagrange remainder term with parallel execution (1: on, 0: off, optional).
- `lagrangeRem.method` – method used to evaluate the Lagrange remainder term ('interval', 'taylorModel' or 'zoo', optional).

Nonlinear System (Lagrange Remainder)

- `lagrangeRem.method` – method used to evaluate the Lagrange remainder term ('interval', 'taylorModel' or 'zoo')
- `lagrangeRem.zooMethods` – cell-array containing the range bounding techniques from which the zoo-object is created.
- `lagrangeRem.maxOrder` – maximum polynomial degree of the monomials in the polynomial part of the Taylor model (optional).
- `lagrangeRem.optMethod` – method used to calculate the bounds of the Talyor model objects ('int', 'bnb', 'bnbAdv' or 'linQuad', optional).
- `lagrangeRem.tolerance` – minimum absolute value of the monomial coefficients in the polynomial part of the Taylor model (optional).
- `lagrangeRem.eps` – termination tolerance ϵ for the branch and bound algorithm and the algorithm based on the Linear Dominated Bounder and the Quadratic Fast Bounder (optional).

Hybrid Systems

- `enclosureEnables` – array containing the identifiers of the methods that are used to approximate the intersections with the guard set.
- `isHyperplaneMap` – flag specifying if the guard-mapping technique is used to calculate the intersections with the guards (1: on, 0: off).
- `startLoc` – ID of the location in which the initial set is located.
- `finalLoc` – execution is terminated if this location is reached.
- `timeStepLoc` – cell-array containing the time step for each location of the hybrid system.
- `uLoc` – cell-array containing the constant input for simulations for each location.
- `Uloc` – cell-array containing the uncertain input set $\tilde{\mathcal{U}}_\Delta$ for each location.
- `uLocTrans` – cell-array containing the translation of the uncertain input set $\tilde{\mathcal{U}}_\Delta$ for each location.

13 Unit Tests

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

- `runTestSuite`: This test suite should always be executed after installing CORA or updating MATLAB/CORA/MPT. This test suite runs the basic tests and should be completed after several minutes. This test suite executes all files in the folder `unitTests` whose function name starts with `test_`.
- `runTestSuite_INTLAB`: This test suite compares the interval arithmetic results with those of INTLAB. 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 the interval arithmetic are already included in `runTestSuite`. This test suite executes all files in the folder `unitTests` whose function name starts with `testINTLAB_`.

14 Loading Simulink and SpaceEx Models

A new feature of CORA 2018 is to load SpaceEx models. This not only has the advantage that one can use the SpaceEx model editor to create models for CORA (see Sec. 14.1.2), but also makes it possible to indirectly load Simulink models through the SL2SX converter [51, 52] (see Sec. 14.1.1). We also plan to make the conversion to CORA available within HYST in the future [53]. We first present how to create SpaceEx models and then how one can convert them to CORA models.

14.1 Creating SpaceEx Models

We present two techniques to create SpaceEx models: a) converting Simulink models to SpaceEx and b) creating models using the SpaceEx model editor.

14.1.1 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. 26.

Please note that the SL2SX converter cannot convert any Simulink model to SpaceEx. A detailed description of limitations can be found in [51, 52].

14.1.2 SpaceEx Model Editor

To create SpaceEx models in an editor, one can use the SpaceEx model editor downloadable from
spaceex.imag.fr/download-6.

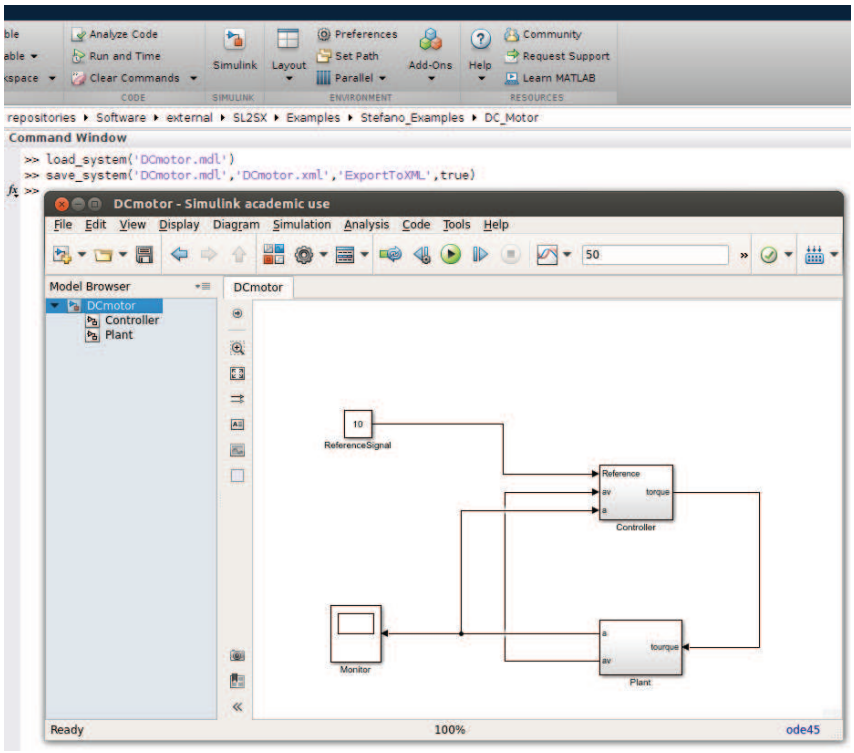


Figure 26: Screenshot of MATLAB/Simulink showing how to save Simulink models in XML format.

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.

A screenshot of the model editor can be found in Fig. 27. Further information on the SpaceEx modeling language is described in [47] and further documents can be downloaded: spaceex.imag.fr/documentation/user-documentation.

Examples of SpaceEx models can be loaded in CORA from `/models/SpaceEx`.

14.2 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 SpaceExModel. 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 model from SpaceEx:

```
1 function HA = bball(~)
2
3
4 %% Generated on 07-Aug-2018
5
```

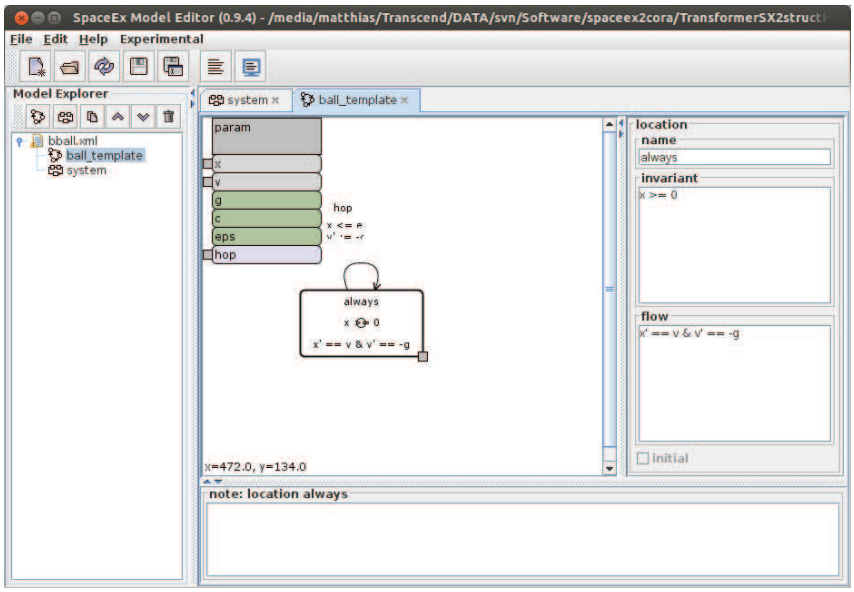


Figure 27: Screenshot of the SpaceEx model editor showing the bouncing ball example.

```
%---------------Automaton created from Component 'system'-----------------

%% Interface Specification:
%   This section clarifies the meaning of state & input 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;-1];
dynamics = linearSys('linearSys', dynA, dynB, dync);

%% equation:
%   x >= 0
invA = ...
[-1,0];
invb = ...
[-0];
invOpt = struct('A', invA, 'b', invb);
inv = mptPolytope(invOpt);

trans = {};
%% equation:
%   v' := -c*v
```

```
resetA = ...
[1,0;0,-0.75];
resetb = ...
[0;0];
reset = struct('A', resetA, 'b', resetb);

%% equation:
%   x <= eps & v < 0
guardA = ...
[1,0;0,1];
guardb = ...
[-0;-0];
guardOpt = struct('A', guardA, 'b', guardb);
guard = mptPolytope(guardOpt);

trans{1} = transition(guard, reset, 1, 'dummy', 'names');

loc{1} = location('S1',1, inv, trans, dynamics);



HA = hybridAutomaton(loc);


end
```

At the beginning of each automatically created model, we list the state 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. 10).

A hand-written version of the bouncing ball example can be found in Sec. 15.2.1 for comparison. How to use the automatically generated bouncing ball model is shown in Sec. 15.2.2.

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. 10) 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.
   - **Invariants & Guards**: CORA requires invariant & guard sets to be modeled by linear inequalities only depending on continuous state variables, resulting in polyhe-

drons in state space. Expressions violating this requirement are ignored and trigger a warning. Furthermore, set definitions must be provided in the format ${a^{(1)}}^T x \leq b^{(1)}$ & ${a^{(2)}}^T x \leq b^{(2)}$ & ... & ${a^{(q)}}^T x \leq b^{(q)}$, where $a^{(i)} \in \mathbb{R}^n$ is a vector, $x \in \mathbb{R}^n$ is the state, and $b^{(i)} \in \mathbb{R}$ is a scalar. The inequalities can also be replaced by equalities to obtain polyhedra which are not full-dimensional. Due to current limitations of our parser even the use of parentheses can cause format errors.

- **Reset Functions**: Resets have to be linear as well and can only depend on the continuous state vector: $x' = Cx + d$, where x' is the state after the reset, $C \in \mathbb{R}^{n \times n}$, $x \in \mathbb{R}^n$ is the state before the reset, and $d \in \mathbb{R}^n$. 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 [54]). Therefore all relevant variables are required to be non-local in all components.
- **Labels**: Synchronization labels (variables of type `label`) are ignored. Neither our parser nor CORA currently implements any synchronized automaton composition.

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');
```

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).

The implementation of the SX2CORA converter is described in detail in Appendix B.

15 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. The examples are 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.

15.1 Continuous Dynamics

15.1.1 Linear Dynamics

For linear dynamics, a simple academic example from [18, Sec. 3.2.3] is used 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 (the function is modified from the original file to better fit in this manual; line numbers after the first line jump due to the removed function description):

```
1  function example_linear_reach_01_5dim()
31
32 dim=5;
33
34 %set options ---------------------------------------------------------------
35 options.tStart=0; %start time
36 options.tFinal=5; %final time
37 options.x0=ones(dim,1); %initial state for simulation
38 options.R0=zonotope([options.x0,0.1*eye(length(options.x0))]); %initial set
39
40 options.timeStep=0.04; %time step size for reachable set computation
41 options.taylorTerms=4; %number of taylor terms for reachable sets
42 options.zonotopeOrder=200; %zonotope order
43 options.originContained=0;
44 options.reductionTechnique='girard';
45
46 uTrans=[1; 0; 0; 0.5; -0.5];
47 options.uTrans=uTrans; %center of uncertain inputs
48 options.U=0.5*zonotope([zeros(5,1),diag([0.2, 0.5, 0.2, 0.5, 0.5])]); %input
49 %---------------------------------------------------------------------------
50
51
52
53 %specify continuous dynamics-------------------------------------------------
54 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];
55 B=1;
56 fiveDimSys=linearSys('fiveDimSys',A,B); %initialize system
```

```
%------------------------------------------------------------------------

%compute reachable set using zonotopes
tic
Rcont = reach(fiveDimSys, options);
tComp = toc;
disp(['computation time of reachable set: ',num2str(tComp)]);

%create random simulations; RRTs would provide better results, but are
%computationally more demanding
runs = 60;
fracV = 0.5;
fracI = 0.5;
changes = 6;
simRes = simulate_random(fiveDimSys, options, runs, fracV, fracI, changes);

%plot results----------------------------------------------------------
for plotRun=1:2
    % plot different projections
    if plotRun==1
        dims=[1 2];
    elseif plotRun==2
        dims=[3 4];
    end

    figure;
    hold on

    %plot reachable sets
    for i=1:length(Rcont)
        plotFilled(Rcont{i},dims,[.8 .8 .8],'EdgeColor','none');
    end

    %plot initial set
    plot(options.R0,dims,'w-','lineWidth',2);

    %plot simulation results
    for i=1:length(simRes.t)
        plot(simRes.x{i}(:,dims(1)),simRes.x{i}(:,dims(2)),'Color',[0 0 0]);
    end

    %abel plot
    xlabel(['x_{',num2str(dims(1)),'}']);
    ylabel(['x_{',num2str(dims(2)),'}']);
end
%------------------------------------------------------------------------
```

The reachable set and the simulation are plotted in Fig. 28 for a time horizon of $t_f = 5$.

15.1.2 Linear Dynamics with Uncertain Parameters

For linear dynamics with uncertain parameters, we use the transmission line example from [21, 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 [21, 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.

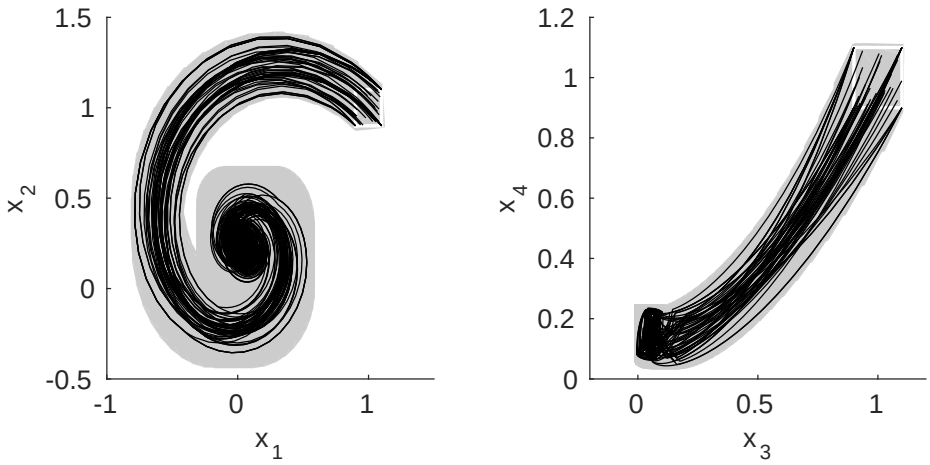


Figure 28: Illustration of the reachable set of the linear example. The white box shows the initial set and the black lines show simulated trajectories.

The MATLAB code that implements the simulation and reachability analysis of the linear example with uncertain parameters is (the function is modified from the original file to better fit in this manual; line numbers after the first line jump due to the removed function description):

```
1  function example_linearParam_reach_01_rlc_const()
32
33 %init: get matrix zonotopes of the model
34 [matZ_A,matZ_B] = initRLC_uTest();
35 matI_A = intervalMatrix(matZ_A);
36
37 %get dimension
38 dim=matZ_A.dim;
39
40 %compute initial set
41 %specify range of voltages
42 u0 = intervalMatrix(0,0.2);
43
44 %compute inverse of A
45 intA = intervalMatrix(matZ_A);
46 invAmid = inv(mid(intA.int));
47
48 %compute initial set
49 intB = intervalMatrix(matZ_B);
50 R0 = invAmid*intB*u0 + intervalMatrix(0,1e-3*ones(dim,1));
51
52 %convert initial set to zonotope
53 R0 = zonotope(interval(R0));
54
55 %initial set
56 options.x0=center(R0); %initial state for simulation
57 options.R0=R0; %initial state for reachability analysis
58
59 %inputs
60 u=intervalMatrix(1,0.01);
61 U = zonotope(interval(intB*u));
62 options.uTrans=center(U);
63 options.U=U+(-options.uTrans); %input for reachability analysis
64
65 %other
```

```
options.tStart=0; %start time
options.tFinal=0.7; %final time
options.intermediateOrder = 2;
options.originContained = 0;
options.timeStep = 0.002;
options.eAt = expm(matZ_A.center*options.timeStep);

options.zonotopeOrder=400; %zonotope order
options.polytopeOrder=3; %polytope order
options.taylorTerms=6;

%time step
r = options.timeStep;
maxOrder=options.taylorTerms;

%instantiate linear dynamics with constant parameters
linSys  = linParamSys(matZ_A, eye(dim), r, maxOrder);
linSys2 = linParamSys(matI_A, eye(dim), r, maxOrder);

%reachable set computations
tic
Rcont = reach(linSys, options);
tComp = toc;
disp(['computation time using matrix zonotopes: ',num2str(tComp)]);

tic
Rcont2 = reach(linSys2, options);
tComp = toc;
disp(['computation time using interval matrices: ',num2str(tComp)]);

%create random simulations; RRTs would provide better results, but are
%computationally more demanding
runs = 60;
fracV = 0.5;
fracI = 0.5;
changes = 6;
simRes = simulate_random(linSys2, options, runs, fracV, fracI, changes);

%plot results--------------------------------------------------------------
for plotRun=1:2
    % plot different projections
    if plotRun==1
        dims=[1 21];
    else
        dims=[20 40];
    end

    figure;
    hold on

    %plot reachable sets
    for i=1:length(Rcont2)
        Zproj = project(Rcont2{i},dims);
        Zproj = reduce(Zproj,'girard',3);
        plotFilled(Zproj,[1 2],[.675 .675 .675],'EdgeColor','none');
    end

    for i=1:length(Rcont)
```

```
124         Zproj = project(Rcont{i},dims);
125         Zproj = reduce(Zproj,'girard',3);
126         plotFilled(Zproj,[1 2],[.75 .75 .75],'EdgeColor','none');
127     end
128 
129     %plot initial set
130     plotFilled(options.R0,dims,'w','EdgeColor','k');
131 
132     %plot simulation results
133     for i=1:length(simRes.t)
134         plot(simRes.x{i}(:,dims(1)),simRes.x{i}(:,dims(2)),'Color',0*[1 1 1]);
135     end
136 
137     %abel plot
138     xlabel(['x_{',num2str(dims(1)),'}']);
139     ylabel(['x_{',num2str(dims(2)),'}']);
140 end
141 
142 %plot results over time
143 
144 figure;
145 hold on
146 
147 %plot time elapse
148 for i=1:length(Rcont2)
149     %get Uout
150     Uout1 = interval(project(Rcont{i},0.5*dim));
151     Uout2 = interval(project(Rcont2{i},0.5*dim));
152     %obtain times
153     t1 = (i-1)*options.timeStep;
154     t2 = i*options.timeStep;
155     %generate plot areas as interval hulls
156     IH1 = interval([t1; infimum(Uout1)], [t2; supremum(Uout1)]);
157     IH2 = interval([t1; infimum(Uout2)], [t2; supremum(Uout2)]);
158 
159     plotFilled(IH2,[1 2],[.675 .675 .675],'EdgeColor','none');
160     plotFilled(IH1,[1 2],[.75 .75 .75],'EdgeColor','none');
161 end
162 
163 %plot simulation results
164 for i=1:(length(simRes.t))
165     plot(simRes.t{i},simRes.x{i}(:,0.5*dim),'Color',[0 0 0]);
166 end
167 
168 %------------------------------------------------------------------------
```

The reachable set and the simulation are plotted in Fig. 29 for a time horizon of $t_f = 0.7$. The plot showing the reachable set of the state x_{20} over time is shown in Fig. 30.

15.1.3 Nonlinear Dynamics

For nonlinear dynamics, several examples are presented.

Tank System The first example is the tank system from [9] where water flows from one tank into another one. This example can be used to study the effect of water power plants on the

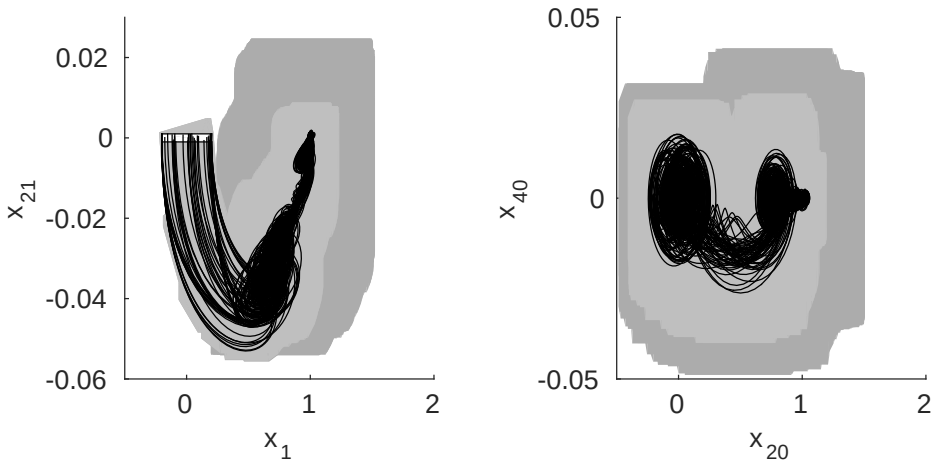


Figure 29: Illustration of the reachable set of the transmission example. The light gray shows the reachable set using matrix zonotopes and the dark gray shows the results using interval matrices. A white box shows the initial set and the black lines are simulated trajectories.

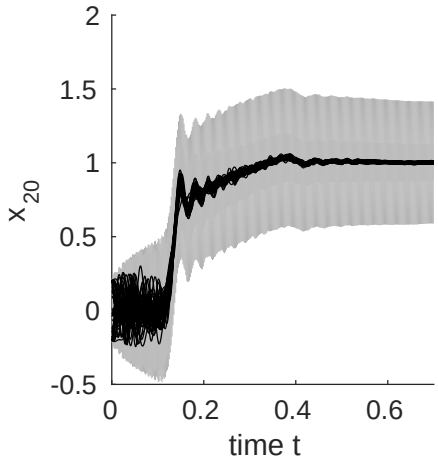


Figure 30: Illustration of the reachable set of the transmission example over time. The light gray shows the reachable set using matrix zonotopes and the dark gray shows the results using interval matrices. Black lines show simulated trajectories.

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 (the function is modified from the original file to better fit in this manual; line numbers after the first line jump due to the removed function description):

```
1  function example_nonlinear_reach_01_tank
37
38  dim=6;
39
40  %set options ---------------------------------------------------------------
41  options.tStart=0; %start time
42  options.tFinal=400; %final time
43  options.x0=[2; 4; 4; 2; 10; 4]; %initial state for simulation
44  options.R0=zonotope([options.x0,0.2*eye(dim)]); %initial set
```

```
45
46  options.timeStep=4; %time step size for reachable set computation
47  options.taylorTerms=4; %number of taylor terms for reachable sets
48  options.zonotopeOrder=50; %zonotope order
49  options.intermediateOrder=5;
50  options.reductionTechnique='girard';
51  options.errorOrder=1;
52  options.polytopeOrder=2; %polytope order
53  options.reductionInterval=1e3;
54  options.maxError = 1*ones(dim,1);
55
56  options.plotType='frame';
57  options.dims=[1 2];
58
59  options.originContained = 0;
60  options.advancedLinErrorComp = 0;
61  options.tensorOrder = 2;
62  %------------------------------------------------------------------------
63
64
65
66
67  %obtain uncertain inputs
68  options.uTrans = 0;
69  options.U = zonotope([0,0.005]); %input for reachability analysis
70
71  %specify continuous dynamics-------------------------------------------
72  tank = nonlinearSys(6,1,@tank6Eq,options); %initialize tank system
73  %------------------------------------------------------------------------
74
75
76  %compute reachable set using zonotopes
77  tic
78  Rcont = reach(tank, options);
79  tComp = toc;
80  disp(['computation time of reachable set: ',num2str(tComp)]);
81
82  %create random simulations; RRTs would provide better results, but are
83  %computationally more demanding
84  runs = 60;
85  fracV = 0.5;
86  fracI = 0.5;
87  changes = 6;
88  simRes = simulate_random(tank, options, runs, fracV, fracI, changes);
89
90  %plot results----------------------------------------------------------
91  for plotRun=1:3
92      % plot different projections
93      if plotRun==1
94          dims=[1 2];
95      elseif plotRun==2
96          dims=[3 4];
97      elseif plotRun==3
98          dims=[5 6];
99      end
100
101     figure;
102     hold on
```

```
103
104     %plot reachable sets
105     for i=1:length(Rcont)
106         plotFilled(Rcont{i}{1},dims,[.8 .8 .8],'EdgeColor','none');
107     end
108
109     %plot initial set
110     plotFilled(options.R0,dims,'w','EdgeColor','k');
111
112     %plot simulation results
113     for i=1:length(simRes.t)
114         plot(simRes.x{i}(:,dims(1)),simRes.x{i}(:,dims(2)),'Color',0*[1 1 1]);
115     end
116
117     %label plot
118     xlabel(['x_{',num2str(dims(1)),'}']);
119     ylabel(['x_{',num2str(dims(2)),'}']);
120 end
121 %------------------------------------------------------------------------
```

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:

```
1  function dx = tank6Eq(t,x,u)
2
3  %parameters
4  k = 0.015;
5  k2 = 0.01;
6  g = 9.81;
7
8  %differential equations
9  dx(1,1)=u(1)+0.1+k2*(4-x(6))-k*sqrt(2*g)*sqrt(x(1)); %tank 1
10 dx(2,1)=k*sqrt(2*g)*(sqrt(x(1))-sqrt(x(2))); %tank 2
11 dx(3,1)=k*sqrt(2*g)*(sqrt(x(2))-sqrt(x(3))); %tank 3
12 dx(4,1)=k*sqrt(2*g)*(sqrt(x(3))-sqrt(x(4))); %tank 4
13 dx(5,1)=k*sqrt(2*g)*(sqrt(x(4))-sqrt(x(5))); %tank 5
14 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. 31 shows the reachable set and the simulation for a time horizon of $t_f = 0.7$.

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 [9] to demonstrate that one can obtain a solution even if the linearization error becomes too large by splitting the reachable set. Later, in [24] an improved method is presented that requires less splitting. This example demonstrates the capabilities of the simpler approach presented in [9]. Due to the similarity of the MATLAB code compared to the previous tank example, we only present the reachable set in Fig. 32.

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*

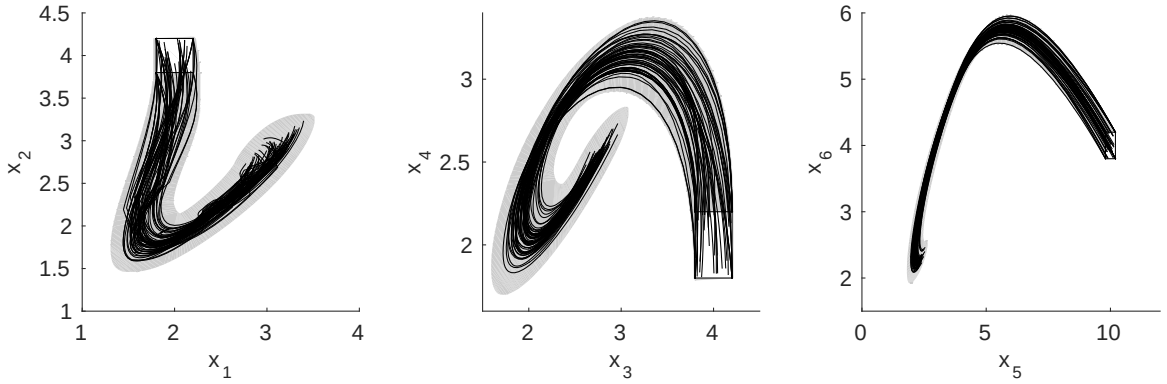


Figure 31: Illustration of the reachable set of the linear example. The white box shows the initial set and the black lines show simulated trajectories.

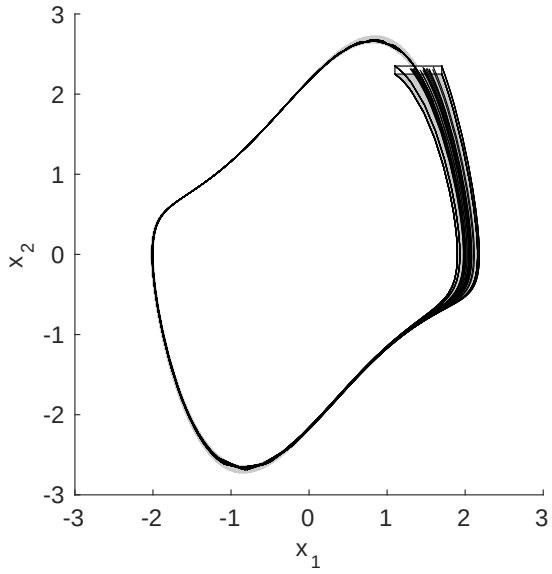


Figure 32: Illustration of the reachable set of the Van der Pol oscillator. The white box shows the initial set and the black lines show simulated trajectories.

is used as presented in [24]. 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. 33.

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., [38,55,56]. 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. 34, where the reference trajectory is plotted in red.

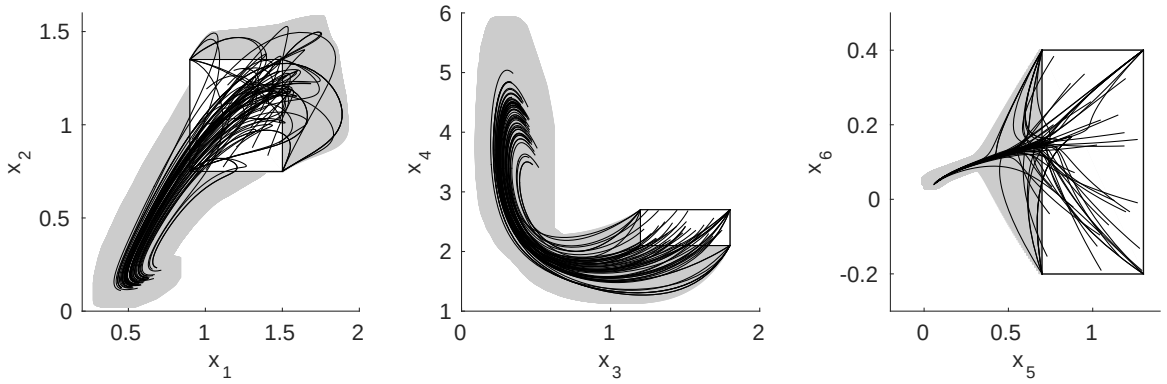


Figure 33: Illustration of the reachable set of the seven-dimensional example for non-convex set representation. The white box shows the initial set and the black lines show simulated trajectories.

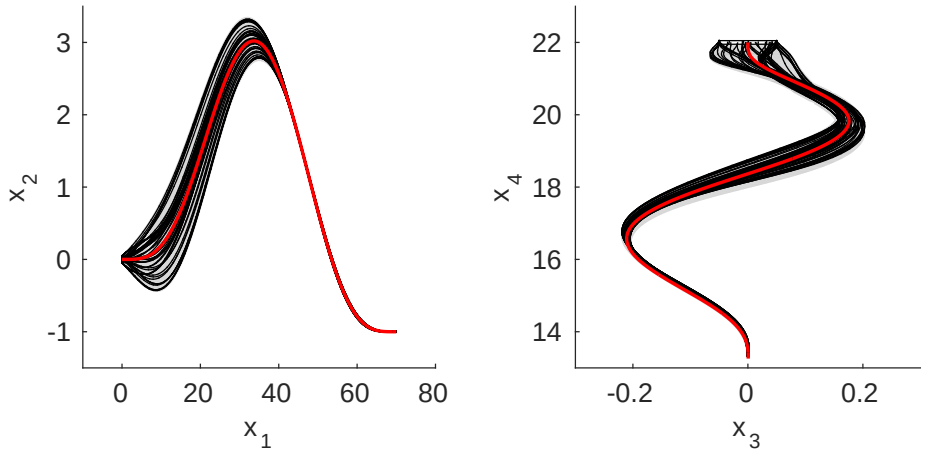


Figure 34: Illustration of the reachable set of the seven-dimensional example for non-convex set representation. The white box shows the initial set and the black lines show simulated trajectories.

15.1.4 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 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 (the function is modified from the original file to better fit in this manual; line numbers after the first line jump due to the removed function description):

```
1  function example_nonlinearParam_reach_01_tank()
37
38 dim=6;
39
40 %set options ---------------------------------------------------------------
41 options.tStart=0; %start time
42 options.tFinal=400; %final time
43 options.x0=[2; 4; 4; 2; 10; 4]; %initial state for simulation
```

```
options.R0=zonotope([options.x0,0.2*eye(dim)]); %initial set
options.timeStep=4;
options.taylorTerms=4; %number of taylor terms for reachable sets
options.intermediateOrder = options.taylorTerms;
options.zonotopeOrder=10; %zonotope order
options.reductionTechnique='girard';
options.maxError = 1*ones(dim,1);
options.reductionInterval=1e3;
options.tensorOrder = 1;

options.advancedLinErrorComp = 0;

options.u=0; %input for simulation
options.U=zonotope([0,0.005]); %input for reachability analysis
options.uTrans=0;

options.p=0.015; %parameter values for simulation
options.paramInt=interval(0.0148,0.015); %parameter intervals
%------------------------------------------------------------------------

%------------------------------------------------------------------------

%specify continuous dynamics with and without uncertain parameters---------
tankParam = nonlinParamSys(6,1,1,@tank6paramEq,options.maxError,options);
tank = nonlinearSys(6,1,@tank6Eq,options);
%------------------------------------------------------------------------

%compute reachable set of tank system with and without uncertain parameters
tic
RcontParam = reach(tankParam,options); %with uncertain parameters
tComp = toc;
disp(['time of reachable set with uncertain parameters: ',num2str(tComp)]);
tic
RcontNoParam = reach(tank, options); %without uncertain parameters
tComp = toc;
disp(['time of reachable set without uncertain parameters: ',num2str(tComp)]);

%create random simulations; RRTs would provide better results, but are
%computationally more demanding
runs = 60;
fracV = 0.5;
fracI = 0.5;
changes = 6;
simRes = simulate_random(tank, options, runs, fracV, fracI, changes);


%plot results----------------------------------------------------------------
plotOrder = 8;
for plotRun=1:3
    % plot different projections
    if plotRun==1
        dims=[1 2];
    elseif plotRun==2
        dims=[3 4];
    elseif plotRun==3
        dims=[5 6];
    end

```

```
    figure;
    hold on

    %plot reachable sets of zonotope; uncertain parameters
    for i=1:length(RcontParam)
        for j=1:length(RcontParam{i})
            Zproj = reduce(RcontParam{i}{j},'girard',plotOrder);
            plotFilled(Zproj,dims,[.675 .675 .675],'EdgeColor','none');
        end
    end

    %plot reachable sets of zonotope; without uncertain parameters
    for i=1:length(RcontNoParam)
        for j=1:length(RcontNoParam{i})
            Zproj = reduce(RcontNoParam{i}{j},'girard',plotOrder);
            plotFilled(Zproj,dims,'w','EdgeColor','k');
        end
    end

    %plot initial set
    plotFilled(options.R0,dims,'w','EdgeColor','k');


    %plot simulation results
    for i=1:length(simRes.x)
        plot(simRes.x{i}(:,dims(1)),simRes.x{i}(:,dims(2)),'k');
    end

    %label plot
    xlabel(['x_{',num2str(dims(1)),'}']);
    ylabel(['x_{',num2str(dims(2)),'}']);
end
%------------------------------------------------------------------------
```

The reachable set and the simulation are plotted in Fig. 35 for a time horizon of $t_f = 400$.

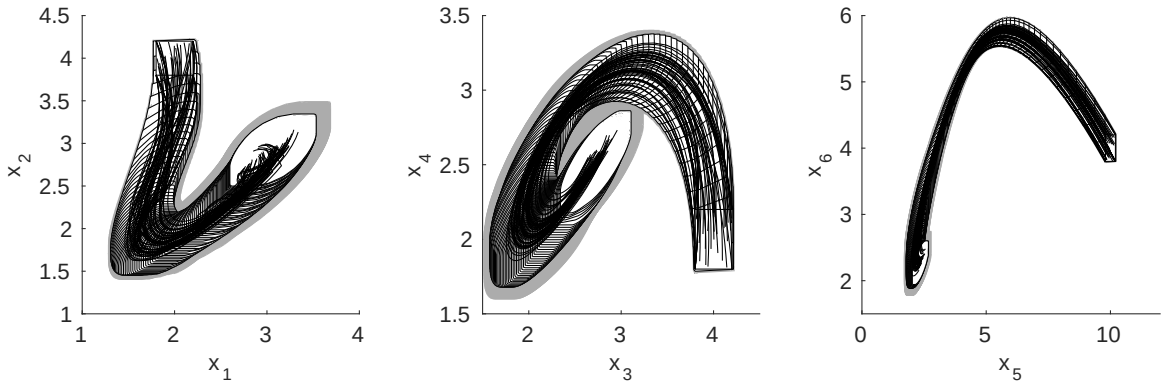


Figure 35: Illustration of the reachable set of the linear example. The gray region shows the reachable set with uncertain parameters, while the white area shows the reachable set without uncertain parameters. Another white box shows the initial set and the black lines show simulated trajectories.

15.1.5 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 [57]. Using the trapezoidal rule for time discretization, we obtained the following nonlinear discrete time system:

$$C_A(k+1) = \frac{1 - \frac{q\tau}{2V} - k_0 \cdot \tau \cdot \exp\left(-\frac{E}{R \cdot T(k)}\right) \cdot C_A(k) + \frac{q}{V} \cdot C_{Af} \cdot \tau}{1 + \frac{q\tau}{2V} + w_1(k) \cdot \tau}$$

$$T(k+1) = \frac{T(k) \cdot \left(1 - \frac{\tau}{2} - \frac{\tau \cdot UA}{2V \cdot \rho \cdot C_p}\right) + \tau \cdot \left(T_f \cdot \frac{q}{V} + \frac{UA \cdot u(C_A(k), T(k))}{V \cdot \rho \cdot C_p}\right)}{1 + \frac{\tau \cdot q}{2V} + \frac{\tau \cdot UA}{2V \cdot \rho \cdot C_p}} \tag{20}$$

$$- \frac{C_A(k) \cdot \frac{\Delta H \cdot k_0 \cdot \tau}{\rho \cdot C_p} \cdot \exp\left(-\frac{E}{R \cdot T(k)}\right)}{1 + \frac{\tau \cdot q}{2V} + \frac{\tau \cdot UA}{2V \cdot \rho \cdot C_p}} + \tau \cdot w_2(k) \quad ,$$

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 τ is the time step size. The values for the model parameters are given in [57]. The MATLAB code that implements the simulation and reachability analysis for the nonlinear discrete time model is shown below:

```
function completed = example_nonlinearDT_reach_cstrDisc

% set options ---------------------------------------------------------------
options.tStart=0;           % start time
options.tFinal=0.15;        % final time
options.x0=[-0.15;-45];     % initial state for simulation
options.R0=zonotope([options.x0,diag([0.005;3])]); % initial set

options.zonotopeOrder=100;              % maximum zonotope order
options.reachabilitySteps=10;           % number of reachability steps
options.timeStep=options.tFinal...
        / options.reachabilitySteps;    % time step size

% additional parameters for reachability analysis
options.tensorOrder = 3;
options.errorOrder = 5;
options.reductionTechnique='girard';


% obtain uncertain inputs ---------------------------------------------------
options.uTrans = [0;0];
options.U = zonotope([zeros(2,1),diag([0.1;2])]); % uncertain inputs


% specify discrete dynamics -------------------------------------------------
sysDisc = nonlinearSysDT(2,2,@cstrDiscr,options);


% compute reachable sets ----------------------------------------------------
R = reach(sysDisc,options);


```

```
% simulate the system ------------------------------------------------------
fractionVertices = 0.5;
fractionInputVertices = 0.5;
runs = 100;

simRes = simulate_random(sysDisc, options, runs, ...
                         fractionVertices, fractionInputVertices);


% plot reachable sets -------------------------------------------------------
hold on
for i=1:length(R)
    Zproj = reduce(R{i},'girard',3);
    plotFilled(Zproj,[1 2],[.8 .8 .8],'EdgeColor','none');
end

% plot simulated trajectories -----------------------------------------------
for i=1:length(simRes.x)
    plot(simRes.x{i}(1,:),simRes.x{i}(2,:),'.k');
end

% add labels to plot --------------------------------------------------------
xlabel('T-T_0');
ylabel('C-C_0');
box on

completed = 1;
%------------------------------------------------------------------------------
```

The reachable set and the simulation are displayed in Fig. 36 for a time horizon of $t_f = 0.15$ min.

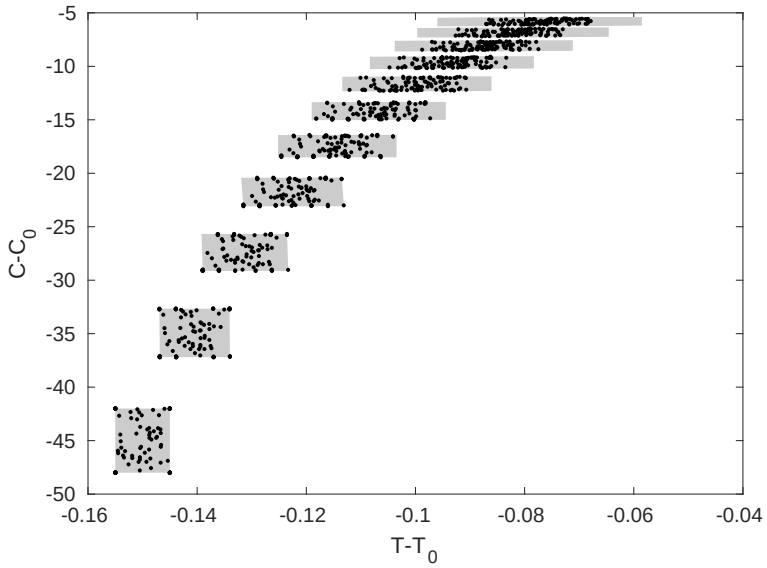


Figure 36: Illustration of the reachable set of the nonlinear discrete-time example. The black dots show the simulated points.

15.1.6 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 [19, 58, 59].

The MATLAB code that implements the simulation and reachability analysis of the nonlinear example with uncertain parameters is (the function is modified from the original file to better fit in this manual; line numbers after the first line jump due to the removed function description):

```
1  function example_nonlinearDA_reach_01_powerSystem_3bus()
26
27 %set path
28 options.tensorOrder = 1;
29
30
31 %specify continuous dynamics-------------------------------------------------
32 powerDyn = nonlinDASys(2,6,2,@bus3Dyn,@bus3Con,options);
33 %-----------------------------------------------------------------------------
34
35 %set options ----------------------------------------------------------------
36 options.tStart = 0; %start time
37 options.tFinal = 5; %final time
38 options.x0 = [380; 0.7]; %initial state
39 options.y0guess = [ones(0.5*powerDyn.nrOfConstraints, 1);
             zeros(0.5*powerDyn.nrOfConstraints, 1)];
40 options.R0 = zonotope([options.x0,diag([0.1, 0.01])]); %initial set
41 options.uTrans = [1; 0.4];
42 options.U = zonotope([zeros(2,1),diag([0, 0.1*options.uTrans(2)])]);
43
44 %options.timeStep=0.01; %time step size for reachable set computation
45 options.timeStep=0.05; %time step size for reachable set computation
46 options.taylorTerms=6; %number of taylor terms for reachable sets
47 options.zonotopeOrder=200; %zonotope order
48 options.errorOrder=1.5;
49 options.polytopeOrder=2; %polytope order
50 options.reductionTechnique='girard';
51
52 options.originContained = 0;
53 options.reductionInterval = 1e5;
54 options.advancedLinErrorComp = 0;
55
56 options.maxError = [0.5; 0];
57 options.maxError_x = options.maxError;
58 options.maxError_y = 0.005*[1; 1; 1; 1; 1; 1];
59 %-----------------------------------------------------------------------------
60
61 %compute reachable set
62 tic
63 Rcont = reach(powerDyn, options);
64 tComp = toc;
65 disp(['computation time of reachable set: ',num2str(tComp)]);
66
67 %create random simulations; RRTs would provide better results, but are
68 %computationally more demanding
```

```
runs = 60;
fracV = 0.5;
fracI = 0.5;
changes = 6;
simRes = simulate_random(powerDyn, options, runs, fracV, fracI, changes);

%plot results----------------------------------------------------------------
dims=[1 2];

figure;
hold on

%plot reachable sets
for i=1:length(Rcont)
    for j=1:length(Rcont{1})
        Zproj = project(Rcont{i}{j},dims);
        Zproj = reduce(Zproj,'girard',3);
        plotFilled(Zproj,[1 2],[.75 .75 .75],'EdgeColor','none');
    end
end

%plot initial set
plotFilled(options.R0,dims,'w','EdgeColor','k');

%plot simulation results
for i=1:length(simRes.t)
    plot(simRes.x{i}(:,dims(1)),simRes.x{i}(:,dims(2)),'Color',0*[1 1 1]);
end

%label plot
xlabel(['x_{',num2str(dims(1)),'}']);
ylabel(['x_{',num2str(dims(2)),'}']);
%----------------------------------------------------------------------------
```

The reachable set and the simulation are plotted in Fig. 37 for a time horizon of $t_f = 5$.

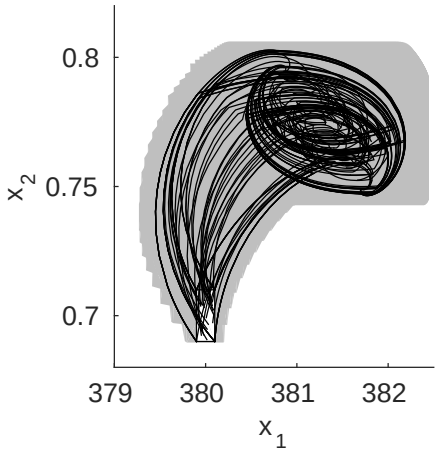


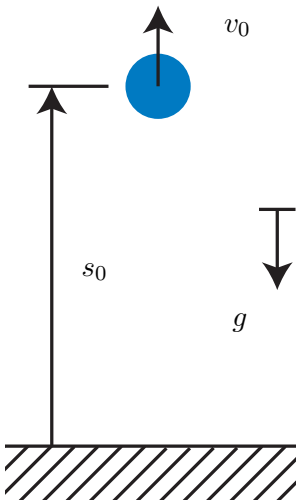
Figure 37: Illustration of the reachable set of nonlinear differential-algebraic example. The white box shows the initial set and the black lines show simulated trajectories.

15.2 Hybrid Dynamics

As already described in Sec. 10, 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. Two major methods are demonstrated by the bouncing ball example and a powertrain example: geometric-based guard intersection for the bouncing ball example and mapping-based guard intersection for the powertrain example. The geometric-based approach is the dominant method in the literature (see e.g., [8,44,60–64]), but the mapping-based approach has shown great scalability for some examples [45]. Determining advantages and disadvantages of both methods require further research.

15.2.1 Bouncing Ball Example

We demonstrate the syntax of CORA for the well-known bouncing ball example, see e.g., [65, Section 2.2.3]. Given is a ball in Fig. 38 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. 10 as



$$
\begin{aligned}
\mathcal{V} &= \{v_1\} \\
\mathcal{X} &= \mathbb{R}^+ \times \mathbb{R} \quad \text{(ball is above ground)} \\
\mathcal{U} = \mathcal{Y}_c &= \{\} \\
\mathtt{T} &= \{(z_1, z_1)\} \\
\mathtt{inv}(z_1) &= \{[x_1, x_2]^T | x_1 \in \mathbb{R}_0^+, x_2 \in \mathbb{R}\} \\
\mathtt{g}\big((z_1, z_1)\big) &= \{[x_1, x_2]^T | x_1 = 0, x_2 \in \mathbb{R}_0^-\} \\
\mathtt{h}\big((z_1, z_1), x\big) &= \begin{bmatrix} x_1 \\ -\alpha\, x_2 \end{bmatrix} \\
\mathtt{f}(z_1, x) &= \begin{bmatrix} x_2 \\ -g \end{bmatrix}
\end{aligned}
$$

Figure 38: Bouncing ball.

The MATLAB code that implements the simulation and reachability analysis of the bouncing ball example is (the function is modified from the original file to better fit in this manual; line numbers after the first line jump due to the removed function description):

```
1  function example_hybrid_reach_01_bouncingBall
28
29 %set options---------------------------------------------------------------
30 options.x0 = [1; 0]; %initial state
31 options.R0 = zonotope([options.x0, diag([0.05, 0.05])]); %initial set
32 options.startLoc = 1; %initial location
33 options.finalLoc = 0; %0: no final location
34 options.tStart = 0; %start time
35 options.tFinal = 1.7; %final time
36 options.timeStepLoc{1} = 0.05; %time step size
37 options.taylorTerms = 10;
38 options.zonotopeOrder = 20;
39 options.polytopeOrder = 10;
```

```
options.errorOrder=2;
options.reductionTechnique = 'girard';
options.isHyperplaneMap = 0;
options.enclosureEnables = 5; %choose enclosure method(s)
options.originContained = 0;
%------------------------------------------------------------------------


%specify hybrid automaton-----------------------------------------------
%specify linear system of bouncing ball
A = [0 1; 0 0];
B = eye(2); % no loss of generality to specify B as the identity matrix
linSys = linearSys('linearSys',A,B);

%define large and small distance
dist = 1e3;
eps = 1e-6;
alpha = -0.75; %rebound factor

%invariant
inv = interval([-2*eps; -dist], [dist; dist]);
%guard sets
guard = interval([-eps; -dist], [0; -eps]);
%resets
reset.A = [0, 0; 0, alpha]; reset.b = zeros(2,1);
%transitions
trans{1} = transition(guard,reset,1,'a','b'); %--> next loc: 1
%specify location
loc{1} = location('loc1',1,inv,trans,linSys);
%specify hybrid automata
HA = hybridAutomaton(loc); % for "geometric intersection"
%------------------------------------------------------------------------

%set input:
options.uLoc{1} = [0; -9.81]; %input for simulation
options.uLocTrans{1} = options.uLoc{1}; %input center
options.Uloc{1} = zonotope(zeros(2,1)); %input deviation

%simulate hybrid automaton
HA = simulate(HA,options);

%compute reachable set
[HA] = reach(HA,options);

%choose projection and plot---------------------------------------------
figure
hold on
options.projectedDimensions = [1 2];
options.plotType = 'b';
plot(HA,'reachableSet',options); %plot reachable set
plotFilled(options.R0,options.projectedDimensions,'w','EdgeColor','k');
plot(HA,'simulation',options); %plot simulation
axis([0,1.2,-6,4]);
%------------------------------------------------------------------------
```

The reachable set and the simulation are plotted in Fig. 39 for a time horizon of $t_f = 1.7$.

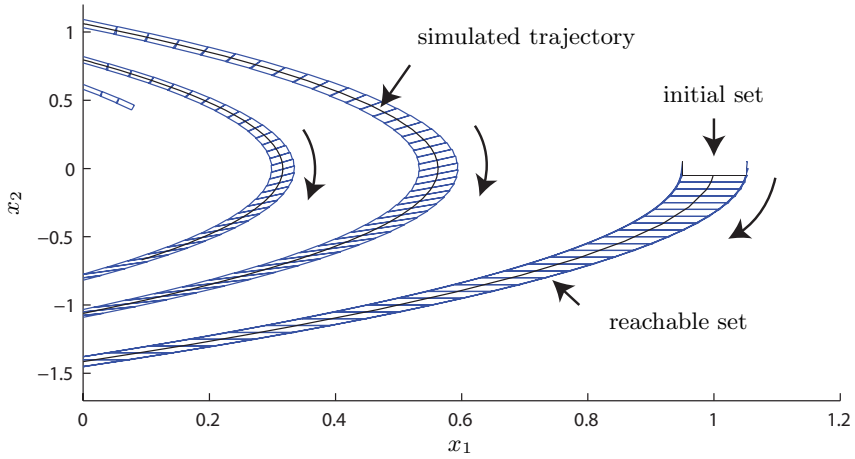


Figure 39: Illustration of the reachable set of the bouncing ball. The black box shows the initial set and the black line shows the simulated trajectory.

15.2.2 Bouncing Ball Example (Converted From SpaceEx)

This example is identical to the bouncing ball example shown in Sec. 15.2.1, except that we use a model that has been automatically converted from SpaceEx. The MATLAB code that implements the simulation and reachability analysis of the bouncing ball example is (the function is modified from the original file to better fit in this manual; line numbers after the first line jump due to the removed function description):

```
1  function completed = example_hybrid_reach_01_bouncingBall_converted
26
27 %set options---------------------------------------------------------------
28 options.x0 = [1; 0]; %initial state for simulation
29 options.R0 = zonotope([options.x0, diag([0.05, 0.05])]); %initial set
30 options.startLoc = 1; %initial location
31 options.finalLoc = 0; %0: no final location
32 options.tStart = 0; %start time
33 options.tFinal = 1.7; %final time
34 options.timeStepLoc{1} = 0.05; %time step size
35 options.taylorTerms = 10;
36 options.zonotopeOrder = 20;
37 options.polytopeOrder = 10;
38 options.errorOrder=2;
39 options.reductionTechnique = 'girard';
40 options.isHyperplaneMap = 0;
41 options.guardIntersect = 'polytope';
42 options.enclosureEnables = 5; %choose enclosure method(s)
43 options.originContained = 0;
44 %------------------------------------------------------------------------
45
46
47 %specify hybrid automaton-----------------------------------------------
48 % converetd hybrid automaton model of the bouncing ball obtained from
49 % "spaceex2cora(bball.xml);"
50 HA = bball;
51 %------------------------------------------------------------------------
52
53 %set input:
54 options.uLoc{1} = 0; % no inputs
55 options.uLocTrans{1} = 0; % no inputs
```

```
options.Uloc{1} = zonotope(0); % no inputs

%simulate hybrid automaton
HA = simulate(HA,options);

%compute reachable set
[HA] = reach(HA,options);

%choose projection and plot---------------------------------------------
figure
hold on
options.projectedDimensions = [1 2];
options.plotType = 'b';
plot(HA,'reachableSet',options); %plot reachable set
plotFilled(options.R0,options.projectedDimensions,'w','EdgeColor','k');
plot(HA,'simulation',options); %plot simulation
axis([0,1.2,-6,4]);
%-----------------------------------------------------------------------
```

The reachable set and the simulation are identical to the model in Sec. 15.2.1 and can be found in Fig. 39 for a time horizon of $t_f = 1.7$.

15.2.3 Powertrain Example

The powertrain example is taken out of [45, 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 plotted in Fig. 40 for a time horizon of $t_f = 2$.

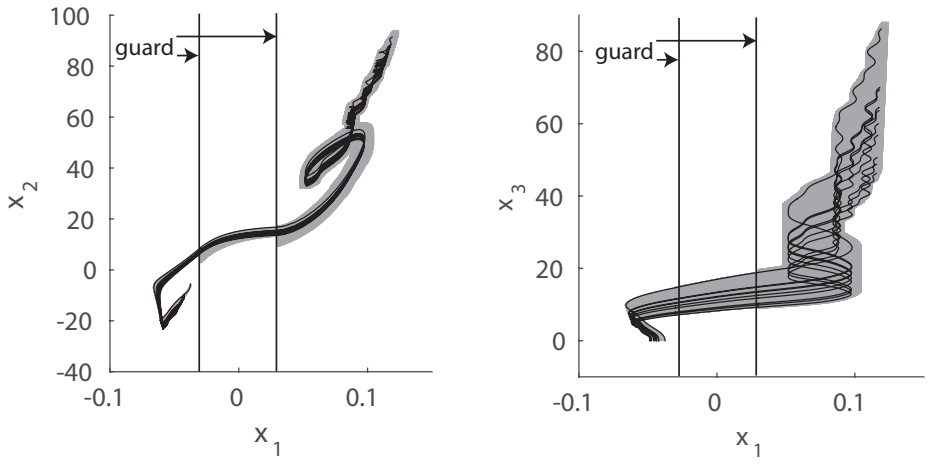


Figure 40: Illustration of the reachable set of the bouncing ball. The black box shows the initial set and the black line shows the simulated trajectory.

16 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 [59]. The efficiency of the algorithms used means it is even possible to verify problems online, i.e., while they are in operation [56].

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 Commission project UnCoVerCPS under grant number 643921.

A Migrating the old `partition` Class into the new one

This table should help to automatically rename old functions to make one's own code compatible with CORA 2018. Details on the functionality of each method can be found in Sec. 11.1.

old command	**new command**
`allSegmentIntervalHulls`	`segmentInterval` (use one input)
`cellCandidates`	`intersectingCells`
`cellCenter`	`cellCenter`
`cellIndices`	`cellIndices`
`cellIntersection`	`intersectingCells`
`cellIntersection2`	`intersectingCells`
`cellIntersection3`	`intersectingCells`
`cellSegments`	`cellSegments`
`centerSegment`	not continued
`display`	`display`
`findSegment`	`intersectingCells`
`findSegments`	`intersectingCells`
`get`	now public
`nextSegment`	not continued
`normalize`	not continued
`nrOfStates`	`nrOfCells`
`plot`	`plot`
`plotHisto`	not continued
`plotHistobars`	not continued
`segmentIntervals`	`cellIntervals`
`segmentPolytope`	`cellPolytopes`
`segmentZonotope`	`cellZonotopes`
`get`	not continued

B Implementation of Loading SpaceEx Models

This section describes the implementation details of the spaceex2cora converter. We will first briefly describe the SpaceEx format in Sec. B.1, followed by an overview of the conversion in Sec. B.2. Details of the conversion are presented in Sec. B.3 and B.4.

B.1 The SpaceEx Format

The SpaceEx format [54] has similarities to statecharts [66]. 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. 10) 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 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[9].

B.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. 14.2.

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

```
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. 15.2.2.

[9] http://spaceex.imag.fr/sites/default/files/spaceex_modeling_language_0.pdf

B.3 Parsing the SpaceEx Components (Phase 1)

Parsing the SpaceEx components is performed in five steps:

1. Accessing XML files (Sec. B.3.1);
2. Parsing component templates (Sec. B.3.2);
3. Building component instances (Sec. B.3.3);
4. Merging component instances (Sec. B.3.4);
5. Conversion to state-space form (Sec. B.3.5).

These steps are described in detail subsequently.

B.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.

B.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. 41.

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.

- id
- **listOfVar(i)**
- **States(i)**
  - name
  - **Flow**
  - **Invariant**
  - **Trans(i)**
    - destination
    - **guard**
    - **reset**

Figure 41: Parsed base component template (indexed fields indicate struct arrays).

As a result, network components are reformatted into the format shown in Fig. 42.

- id
- **listOfVar(i)**
- **Binds(i)**
  - id
  - keys
  - values
  - values_text

Figure 42: Parsed network component template (indexed fields indicate struct arrays).

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+` 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.

B.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. 43. 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 the step.

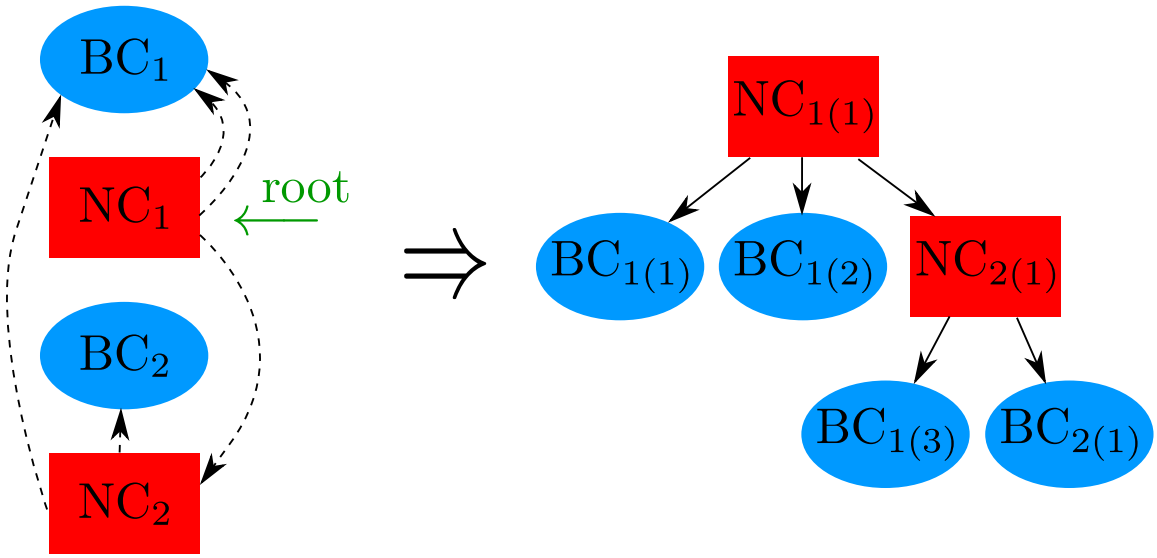


Figure 43: Example for the composition of the component tree. The red nodes represent Network components (NC), and the blue nodes base components (BC). Dashed arrows depict references, while solid arrows represent instantiations.

B.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. 43, the state vector could for example look as follows:

$$\vec{x} = (\underbrace{x_1,\ x_2}_{BC_{1(1)}},\ \underbrace{x_3,\ x_4}_{BC_{1(2)}},\ \underbrace{x_5,\ x_6}_{BC_{1(3)}},\ \underbrace{x_7,\ x_8,\ x_9}_{BC_{2(1)}})^T \tag{21}$$

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 analysis of Compositional Hybrid Automata (see Sec. 10.5). 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 in-

stances 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 [67, Chapter 5] as well as in [46, Def. 2.9].

B.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. In the following, 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 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 Analogously to linear flow functions, reset functions $r(x)$ are evaluated to obtain the form $r(x) = Ax + b$. A failure of the linearity test causes an error here, since CORA currently does not support nonlinear reset functions.

Guard Sets and Invariant Sets The SpaceEx modeling language uses polyhedra for continuous sets. CORA can store polyhedra with the class `mptPolytope`, which is based on the `Polyhedron` class of the Multi-Parametric Toolbox 3 for MATLAB[10].

Polyhedra can be specified by the coefficients $C \in \mathbb{R}^{p \times n}$, $d \in \mathbb{R}^p$, $C_e \in \mathbb{R}^{q \times n}$, $d_e \in \mathbb{R}^q$ 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 $C_e x = d_e$

[10] people.ee.ethz.ch/ mpt/3/

are obtained via partial derivatives and insertion of zeros. Nonlinearity causes an error, since only linear sets are supported by CORA.

B.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 currently not support zero-input automata. For this reason we have added a dummy input without any in the example above.

B.5 Open Problems

The spaceex2cora converter has already been used in the ARCH 2018 friendly competition. However, its development is far from being finished. We suggest addressing the following issues in the future:

- **Input constraints:** Input constraints are in the SpaceEx format specified as 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. We plan for the future to automatically create `linParamSys` or `nonlinParamSys` objects if uncertain system parameters are present.
- **Synchronized composition:** The SpaceEx format enables the creation of synchronized hybrid automata. Since CORA currently does not support synchronization, it would be good to implement this functionality in CORA.

C Licensing

CORA is released under the GPLv3.

D 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 `http://www6.in.tum.de/`.

E Contributors

All people that have contributed so far are listed in alphabetical order of the last name in Tab. 10. The table further shows the number of files for each of the different CORA modules that an auther contributed to.

Table 10: Number of files that an author contributed to, patitioned by the different modules of CORA.

	contDynamics	**contSet**	**discrDynamics**	**evaluations**	**examples**	**global functions**	**hybridDynamics**	**matrixSet**	**models**	**spaceex2cora**	**testFunctions**	**unitTests**
Daniel Althoff	-	1	-	-	-	-	-	-	-	-	-	2
Matthias Althoff	150	369	92	-	21	34	59	69	4	3	35	68
Victor Charlent	-	1	-	-	-	-	3	-	-	-	-	-
Changshun Deng	-	-	-	-	-	1	-	-	-	-	-	-
Ahmed El-Guindy	-	-	-	-	2	-	-	-	-	-	-	-
Dmitry Grebenyuk	-	67	-	-	-	7	-	-	-	-	-	63
Niklas Kochdumper	22	69	1	-	4	10	21	-	-	1	30	20
Anna Kopetzki	-	2	-	2	-	-	-	-	-	-	-	-
Stefan Liu	-	-	-	-	-	-	1	-	-	-	-	-
Aaron Pereira	-	-	7	-	-	-	-	-	-	-	-	7
Hendrik Röhm	-	-	-	-	-	-	-	-	-	-	-	1
Johann Schöpfer	-	-	-	-	-	-	2	-	-	-	-	-

References

[1] 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, pp. 725–730.

[2] 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, pp. 5626–5633.

[3] 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.

[4] 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, pp. 126–136, 2016.

[5] G. Lafferriere, G. J. Pappas, and S. Yovine, "Symbolic reachability computation for families of linear vector fields," *Symbolic Computation*, vol. 32, pp. 231–253, 2001.

[6] M. Althoff, "An introduction to CORA 2015," in *Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems*, 2015, pp. 120–151.

[7] 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, pp. 91–105.

[8] 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, pp. 379–395.

[9] 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, pp. 4042–4048.

[10] 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, pp. 7439–7446.

[11] 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.

[12] ——, *ARCH18. 5th International Workshop on Applied Verification for Continuous and Hybrid Systems*, ser. EasyChair Proceedings in Computing. EasyChair, 2018.

[13] 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, pp. 143–159.

[14] 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.

[15] 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, pp. 160–169.

[16] 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.

[17] A. Girard, "Reachability of uncertain linear systems using zonotopes," in *Hybrid Systems: Computation and Control*, ser. LNCS 3414. Springer, 2005, pp. 291–305.

[18] 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.

[19] M. Althoff and B. H. Krogh, "Reachability analysis of nonlinear differential-algebraic systems," *IEEE Transactions on Automatic Control*, vol. 59, no. 2, pp. 371–383, 2014.

[20] E. Gover and N. Krikorian, "Determinants and the volumes of parallelotopes and zonotopes," *Linear Algebra and its Applications*, vol. 433, no. 1, pp. 28–40, 2010.

[21] 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, pp. 69–94.

[22] C. Combastel, "A state bounding observer based on zonotopes," in *Proc. of the European Control Conference*, 2003.

[23] 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, pp. 6814–6821.

[24] M. Althoff, "Reachability analysis of nonlinear systems using conservative polynomialization and non-convex sets," in *Hybrid Systems: Computation and Control*, 2013, pp. 173–182.

[25] J. Hoefkens, M. Berz, and K. Makino, *Scientific Computing, Validated Numerics, Interval Methods*. Springer, 2001, ch. Verified High-Order Integration of DAEs and Higher-Order ODEs, pp. 281–292.

[26] 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, pp. 625–630.

[27] D. Berleant, "Automatically verified reasoning with both intervals and probability density functions," *Interval Computations*, vol. 2, pp. 48–70, 1993.

[28] G. M. Ziegler, *Lectures on Polytopes*, ser. Graduate Texts in Mathematics. Springer, 1995.

[29] V. Kaibel and M. E. Pfetsch, *Algebra, Geometry and Software Systems*. Springer, 2003, ch. Some Algorithmic Problems in Polytope Theory, pp. 23–47.

[30] M. Berz and G. Hoffstätter, "Computation and application of Taylor polynomials with interval remainder bounds," *Reliable Computing*, vol. 4, pp. 83–97, 1998.

[31] K. Makino and M. Berz, "Remainder Differential Algebras and Their Applications," 1996. [Online]. Available: http://bt.pa.msu.edu/pub/papers/rdasf/rdasf.pdf

[32] ——, "Taylor models and other validated functional inclusion methods," *International Journal of Pure and Applied Mathematics*, vol. 4, no. 4, pp. 379–456, 2003.

[33] ——, "Rigorous integration of flows and ODEs using Taylor models," in *Proc. of Symbolic-Numeric Computation*, 2009, pp. 79–84.

[34] R. D. Neidinger, "Directions for computing truncated multivariate Taylor series," *Mathematics of Computation*, vol. 74, no. 249, pp. 321–340, 2004.

[35] L. H. de Figueiredo and J. Stolfi, "Affine arithmetic: Concepts and applications," *Numerical Algorithms*, vol. 37, no. 1-4, pp. 147–158, 2004. [Online]. Available: http://link.springer.com/10.1023/B:NUMA.0000049462.70970.b6

[36] W. Kühn, *Mathematical Visualization*. Springer, 1998, ch. Zonotope Dynamics in Numerical Quality Control, pp. 125–134.

[37] O. Stursberg and B. H. Krogh, "Efficient representation and computation of reachable sets for hybrid systems," in *Hybrid Systems: Computation and Control*, ser. LNCS 2623. Springer, 2003, pp. 482–497.

[38] 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, pp. 3559–3566.

[39] M. Althoff, C. Le Guernic, and B. H. Krogh, "Reachable set computation for uncertain time-varying linear systems," in *Hybrid Systems: Computation and Control*, 2011, pp. 93–102.

[40] 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, pp. 257–271.

[41] C. W. Gardiner, *Handbook of Stochastic Methods: For Physics, Chemistry and the Natural Sciences*, H. Haken, Ed. Springer, 1983.

[42] 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.

[43] K. E. Brenan, S. L. Campbell, and L. R. Petzold, *Numerical Solution of Initial Value Problems in Differential-Algebraic Equations*. North-Holland, 1989.

[44] 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, pp. 233–249, 2010.

[45] M. Althoff and B. H. Krogh, "Avoiding geometric intersection operations in reachability analysis of hybrid systems," in *Hybrid Systems: Computation and Control*, 2012, pp. 45–54.

[46] G. Frehse, "Compositional verification of hybrid systems using simulation relations," Ph.D. dissertation, Radboud Universiteit Nijmegen, 2005.

[47] A. Donzé and G. Frehse, "Modular, hierarchical models of control systems in SpaceEx," in *Proc. of the European Control Conference*, 2013, pp. 4244–4251.

[48] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Springer, 2009.

[49] 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, pp. 299 – 310, 2009.

[50] 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, pp. 1237–1247, 2011.

[51] 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, pp. 93–98.

[52] 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, pp. 1788–1795.

[53] 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.

[54] S. Cotton, G. Frehse, and O. Lebeltel. (2010) The spaceex modeling languag. [Online]. Available: http://spaceex.imag.fr/sites/default/files/spaceex_modeling_language_0.pdf

[55] 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, pp. 1162–1167.

[56] ——, "Online verification of automated road vehicles using reachability analysis," *IEEE Transactions on Robotics*, vol. 30, no. 4, pp. 903–918, 2014.

[57] 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, pp. 1745–1751, 2006.

[58] 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, pp. 1–8.

[59] M. Althoff, "Formal and compositional analysis of power systems using reachable sets," *IEEE Transactions on Power Systems*, vol. 29, no. 5, pp. 2270–2280, 2014.

[60] 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, pp. 215–228.

[61] G. Frehse, "PHAVer: Algorithmic verification of hybrid systems past HyTech," *International Journal on Software Tools for Technology Transfer*, vol. 10, pp. 263–279, 2008.

[62] N. Ramdani and N. S. Nedialkov, "Computing reachable sets for uncertain nonlinear hybrid systems using interval constraint-propagation techniques," *Nonlinear Analysis: Hybrid Systems*, vol. 5, no. 2, pp. 149–162, 2010.

[63] G. Frehse and R. Ray, "Flowpipe-guard intersection for reachability computations with support functions," in *Proc. of Analysis and Design of Hybrid Systems*, 2012, pp. 94–101.

[64] X. Chen, "Reachability analysis of non-linear hybrid systems using taylor models," Ph.D. dissertation, RWTH Aachen University, 2015.

[65] A. van der Schaft and H. Schumacher, *An Introduction to Hybrid Dynamical Systems*. Springer, 2000.

[66] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[67] E. A. Lee and S. A. Seshia, *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press, 2016.