CORA 2016 Manual

Matthias Althoff

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 them 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 2015?

Non-exhaustive and unsorted list:

- CORA no longer requires the MATLAB toolbox INTLAB for applying interval arithmetic. This is mainly motivated by the fact that INTLAB is no longer freely available. Please note that the CORA implementation of interval arithmetic does not consider errors caused by finite machine precision. Details of the implementation can be found in [1]. If consideration of machine precision is important, one should purchase INTLAB.
- Introduction of unit tests to better ensure that functionality is maintained after larger software changes. The unit tests can also be used as guiding examples to set up own verification problems. More on the introduced unit tests can be found in Sec. 11.
- It is no longer required to implement all systems as a hybrid automaton in order to use the method *reach* for computing the reachable set. The keyword is now also reserved for reachability analysis of purely continuous systems.
- Auxiliary files, such as the Lagrange remainder now contain the name of the model and are no longer overwritten when changing the investigated model.
- To shorten the code while not compromising functionality, we have integrated the class `intervalhull` into the new class `interval` and the classes `vehicleSys` and `vehicleSys_td` into the existing class *nonlinearSys*.
- Faster plotting of reachable sets thanks to a new routine from Daniel Heß.
- The 2015 version only contained the bouncing ball example. The new version has for each implemented category of dynamical systems at least one example, which are presented in Sec. 12.
- Many unused or prototypical files have been removed and the code has been decluttered for various functions.

2 Philosophy and Architecture

The **CO**ntinuous **R**eachability **A**nalyzer (CORA)[1] is a MATLAB toolbox for prototypical design of algorithms for reachability analysis. The toolbox is designed for various kinds of systems with purely continuous dynamics (linear systems, nonlinear systems, differential-algebraic systems, parameter-varying systems, etc.) and hybrid dynamics combining the aforementioned continuous dynamics with discrete 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 [2] 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: $\mathcal{R}([t_0, t_f]) = \bigcup_{t \in [t_0, t_f]} \mathcal{R}(t)$.

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

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[2], 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. 10). 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].

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 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.

4 Architecture

The architecture of CORA can essentially be grouped into the following parts based on a separation of concerns as presented in Fig. 1 using UML[3]: Classes for set representations (Sec. 5), classes for matrix set representations (Sec. 6), classes for the analysis of continuous dynamics

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

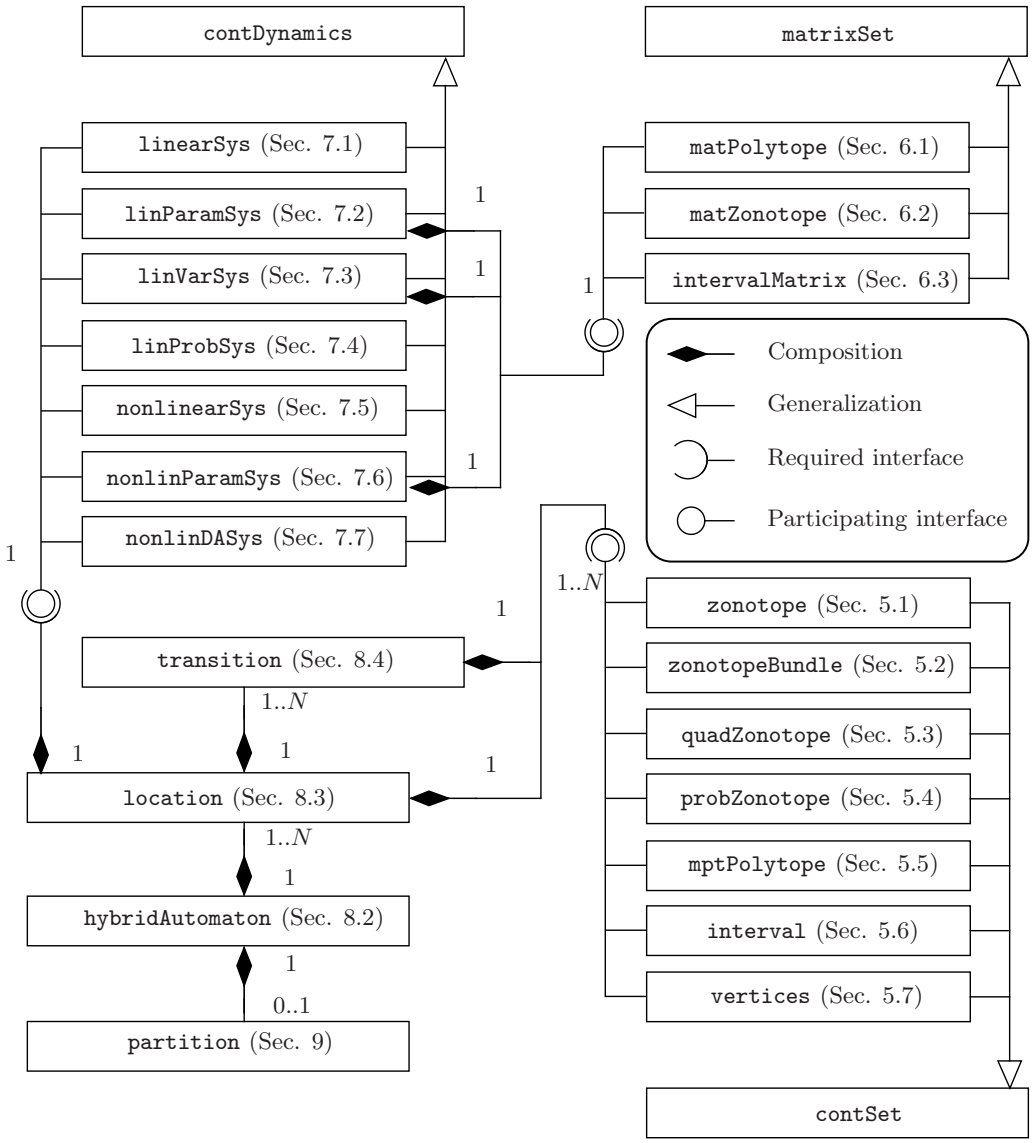


Figure 1: Unified Modeling Language (UML) class diagram of CORA.

(Sec. 7), classes for the analysis of hybrid dynamics (Sec. 8), and a class for the partitioning of the state space (Sec. 9).

The class diagram in Fig. 1 shows that hybrid systems (class `hybridAutomaton`) consists 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. 8.

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.

5 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 as depicted in Fig. 2, which also shows conversions supported between set representations.

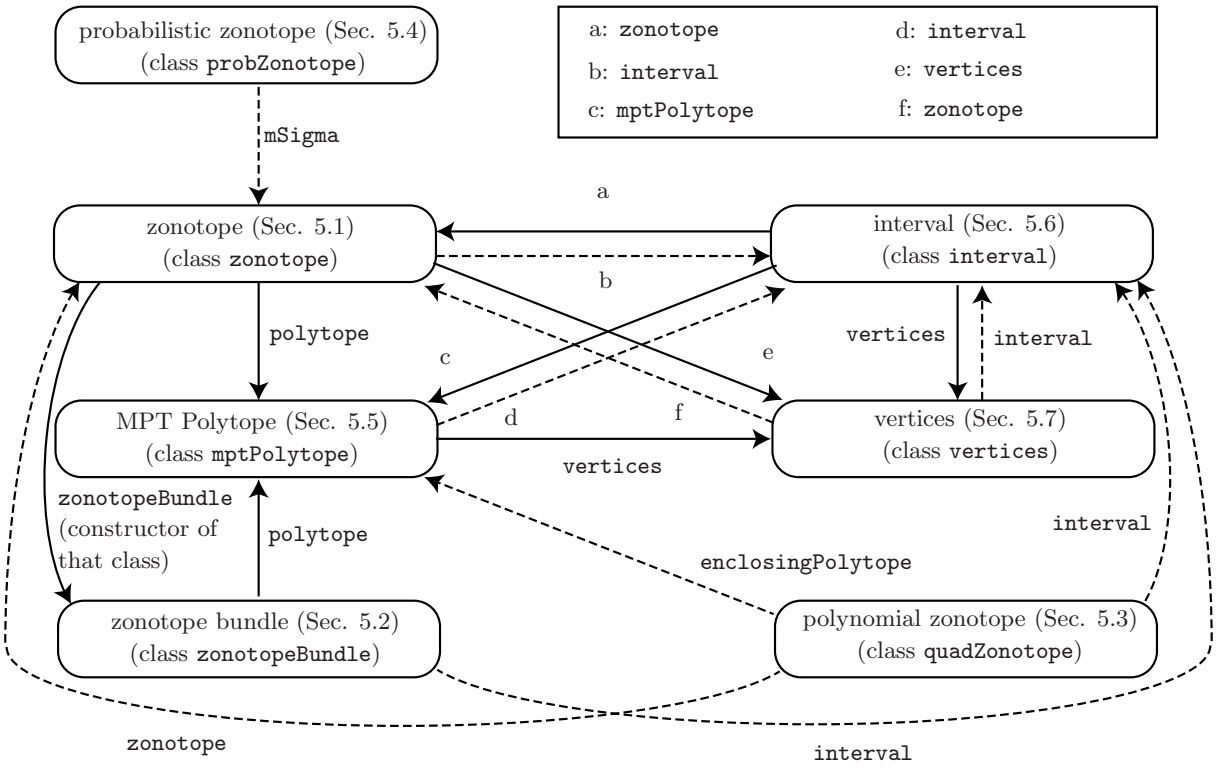


Figure 2: Set conversions supported. Solid arrows represent exact conversions, while dashed arrows represent over-approximative conversions. The arrows are labeled by the corresponding method to carry out the conversion.

Important operations for sets are:

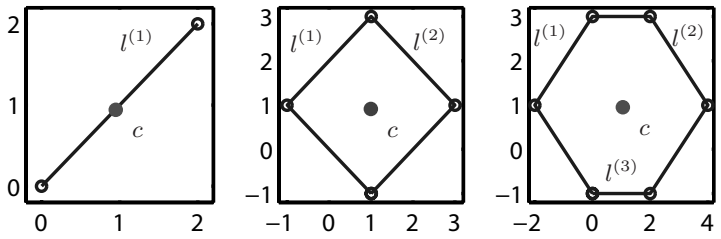
- `display`: Displays the parameters of the set in the MATLAB workspace.
- `plot`: Plots a two-dimensional projection of a set in the current MATLAB figure.
- `mtimes`: Overloads the '`*`' operator for the multiplication of various objects with a set. For instance if M is a matrix of proper dimension 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. 5.6).

5.1 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

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

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. 3. 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 [4].



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

Figure 3: 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.
- `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. 5.
- `enclose` – generates a zonotope that encloses two zonotopes of equal dimension according to [5, 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 [5, Sec. 2.5.6].
- `enlarge` – enlarges the generators of a zonotope by a vector of factors for each dimension.
- `inParallelotope` – checks if a zonotope is a subset of a parallelotope, where the latter is represented as a zonotope.

- `interval` – standard method, see Sec. 5. More details can be found in [5, Proposition 2.2].
- `is_empty` – returns 1 if a zonotope is empty and 0 otherwise.
- `mtimes` – standard method, see Sec. 5. More details can be found in Sec. 5.1.1.
- `plot` – standard method, see Sec. 5. More details can be found in Sec. 5.8.
- `plus` – standard method, see Sec. 5. More details can be found in Sec. 5.1.2.
- `polygon` – converts a two-dimensional zonotope into a polygon and returns its vertices.
- `polytope` – returns an exact polytope in halfspace representation according to [5, 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 [6, Lemma 1].
- `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. 5.1.3.
- `split` – splits a zonotope into two or more zonotopes that enclose the original zonotope. More details can be found in Sec. 5.1.4.
- `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 [7, p.40].
- `zonotope` – constructor of the class.

5.1.1 Method `mtimes`

Table 1 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 1: Classes that can be multiplied with a zonotope.

class	reference	literature
MATLAB matrix	-	-
`interval`	Sec. 5.6	[5, Theorem 3.3]
`intervalMatrix`	Sec. 6.3	[5, Theorem 3.3]
`matZonotope`	Sec. 6.2	[8, Sec. 4.4.1]

5.1.2 Method `plus`

Table 2 lists the classes that can be added to a zonotope. Other than for multiplication, the zonotope can be on both sides of the `'+'` operator.

Table 2: Classes that can be added to a zonotope.

class	reference	literature
MATLAB vector	-	-
`zonotope`	Sec. 5.1	[5, Equation 2.1]

5.1.3 Method `reduce`

The zonotope reduction returns an over-approximating zonotope with less generators as described in [5, Proposition 2.5]. Table 3 lists some of the implemented reduction techniques. The standard reduction technique is `'girard'`.

Table 3: Reduction techniques for zonotopes.

reduction technique	primary use	literature
`girard`	Reduction of high to medium order	[4, Sec. 3.4]
`MethA`	Reduction to parallelotope	Method A in [5, Sec. 2.5.5]
`MethB`	Reduction to parallelotope	Method B in [5, Sec. 2.5.5]
`MethC`	Reduction to parallelotope	Method C in [5, Sec. 2.5.5]

5.1.4 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 4 can be selected to optimize the split performance.

Table 4: Split techniques for zonotopes.

split technique	comment	literature
`splitOneGen`	splits one generator	[5, Proposition 3.8]
`directionSplit`	splits all generators in one direction	—
`directionSplitBundle`	exact split using zonotope bundles	[9, Section V.A]
`halfspaceSplit`	split along a given halfspace	—

5.1.5 Zonotope Example

The following MATLAB code demonstrates some of the introduced methods:

```
1 Z1 = zonotope([1 1 1; 1 -1 1]); % create zonotope Z1
2 Z2 = zonotope([-1 1 0; 1 0 1]); % create zonotope Z2
3 A = [0.5 1; 1 0.5]; % numerical matrix A
```

```
4
5  Z3 = Z1 + Z2; % Minkowski addition
6  Z4 = A*Z3; % linear map
7
8  figure; hold on
9  plot(Z1,[1 2],'b'); % plot Z1 in blue
10 plot(Z2,[1 2],'g'); % plot Z2 in green
11 plot(Z3,[1 2],'r'); % plot Z3 in red
12 plot(Z4,[1 2],'k'); % plot Z4 in black
13
14 P = polytope(Z4) % convert to and display halfspace representation
15 I = interval(Z4) % convert to and display intervah hull
16
17 figure; hold on
18 plot(Z4); % plot Z4
19 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. 4 and the ones generated in lines 18-19 are shown in Fig. 5.

5.2 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 [9]. 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.

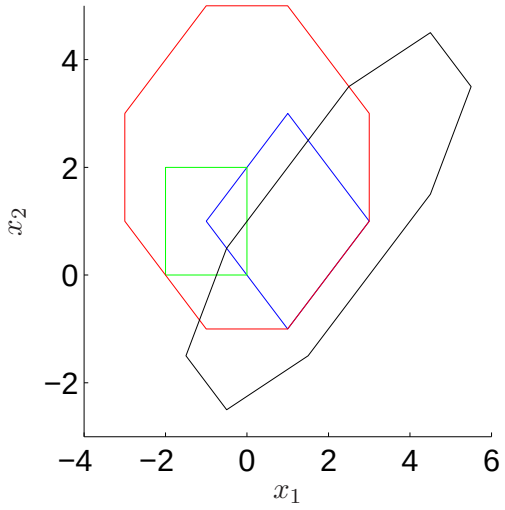


Figure 4: Zonotopes generated in lines 9-12 of the zonotope example in Sec. 5.1.5.

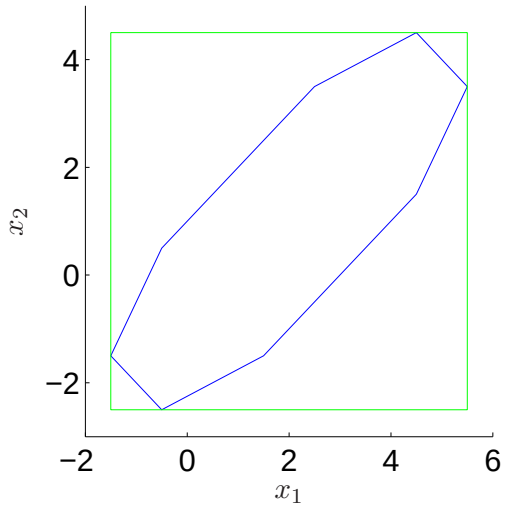


Figure 5: Sets generated in lines 18-19 of the zonotope example in Sec. 5.1.5.

- `display` – standard method, see Sec. 5.
- `enclose` – generates a zonotope bundle that encloses two zonotopes bundles of equal dimension according to [9, Proposition 5].
- `encloseTight` – generates a zonotope bundle that encloses two zonotopes bundles in a possibly tighter way than `enclose` as outlined in [9, Sec. VI.A].
- `enlarge` – enlarges the generators of each zonotope in the bundle by a vector of factors for each dimension.
- `enclosingPolytope` – returns an over-approximating polytope in halfspace representation. For each zonotope the method `enclosingPolytope` of the class `zonotope` in Sec. 5.1 is called.
- `interval` – standard method, see Sec. 5. More details can be found in [9, Proposition 6].
- `mtimes` – standard method, see Sec. 5. More details can be found in [9, Proposition 1].
- `plot` – standard method, see Sec. 5. More details can be found in Sec. 5.8.
- `plus` – standard method, see Sec. 5. More details can be found in [9, Proposition 2].
- `polytope` – returns an exact polytope in halfspace representation. Each zonotope is converted to halfspace representation according to [5, 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.
- `reduce` – returns an over-approximating zonotope bundle with less generators. For each zonotope the method `reduce` of the class `zonotope` in Sec. 5.1 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.
- `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.

5.2.1 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. 6.

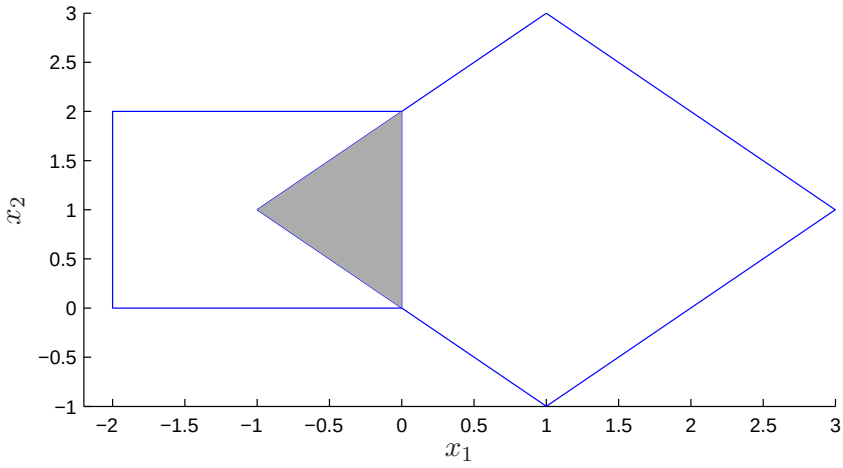


Figure 6: Sets generated in lines 7-9 of the zonotope bundle example in Sec. 5.2.1.

5.3 Polynomial Zonotopes

Zonotopes are a very efficient representation for reachability analysis of linear systems [4] and of nonlinear systems that can be well abstracted by linear differential inclusions [5]. However, more advanced techniques, such as in [10], 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 [10]. 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 [11] (as briefly discussed later) and is a generalization of zonotopes.

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

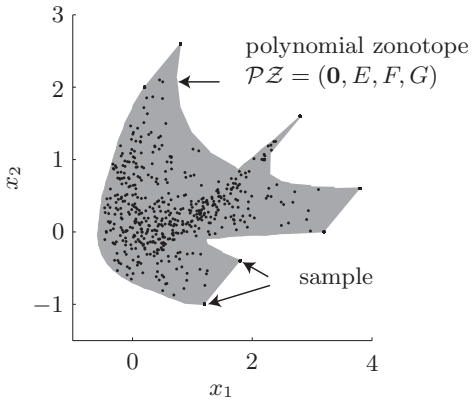
$$\begin{aligned}
E^{[i]} &= [\underbrace{f^{([i],1,1,\ldots,1)}}_{=:e^{([i],1)}} \ldots \underbrace{f^{([i],p,p,\ldots,p)}}_{=:e^{([i],p)}}] \text{ (all indices are the same value)},\\
F^{[i]} &= [f^{([i],1,1,\ldots,1,2)}\ f^{([i],1,1,\ldots,1,3)} \ \ldots\ f^{([i],1,1,\ldots,1,p)}\\
&\quad\ f^{([i],1,1,\ldots,2,2)}\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

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. 7.

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`.



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

Figure 7: 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.

- `center` – returns the starting point c.
- `display` – standard method, see Sec. 5.
- `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. 5. 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. 5 as stated in [9, Equation 14] for numeric matrix multiplication. As described in Sec. 5.1.1 the multiplication of interval matrices is also supported, whereas the implementation for matrix zonotopes is not yet implemented.
- `plot` – standard method, see Sec. 5. More details can be found in Sec. 5.8.
- `plus` – standard method, see Sec. 5. 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 [10, 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. 5.3.1.
- `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 [10, Proposition 1].

5.3.1 Method `reduce`

The zonotope reduction returns an over-approximating zonotope with less generators. Table 5 lists the implemented reduction techniques.

Table 5: Reduction techniques for zonotopes.

reduction technique	**comment**	**literature**
`redistribute`	Changes dependent and independent generators	[10, Proposition 2]
`girard`	Only changes independent generators as for a regular zonotope	[4, Sec. 3.4]

5.3.2 Polynomial Zonotope Example

The following MATLAB code demonstrates some of the introduced methods:

```
c = [0;0]; % starting point
E1 = diag([-1,0.5]); % generators of factors with identical indices
E2 = [1 1; 0.5 0.3]; % generators of factors with identical indices
F = [-0.5; 1]; % generators of factors with different indices
G = [0.3; 0.3]; % independent generators

qZ = quadZonotope(c,E1,E2,F,G); % instantiate quadratic zonotope
Z = zonotope(qZ) % over-approximate by a zonotope

figure; hold on
plot(Z); % plot Z
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. 8.

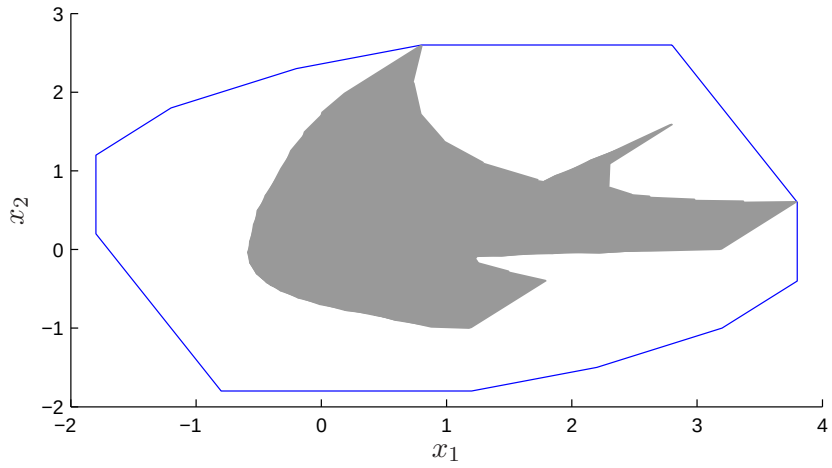


Figure 8: Sets generated in lines 11-12 of the polynomial zonotope example in Sec. 5.3.2.

5.4 Probabilistic Zonotopes

Probabilistic zonotopes have been introduced in [12] 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 [4]. 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: $\mathcal{Z}_g = (c, \underline{g}^{(1\ldots q)})$.

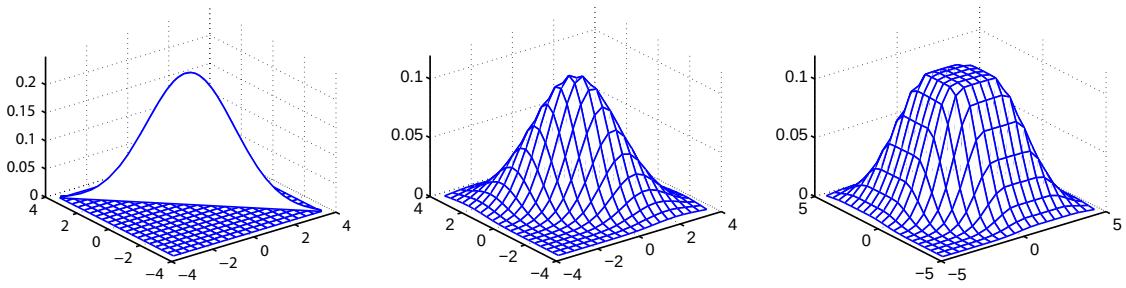
A Gaussian zonotope with uncertain mean $\mathscr{Z}$ is defined as a Gaussian zonotope $\mathcal{Z}_g$, where the center is uncertain and can have any value within a zonotope $\mathcal{Z}$, which is denoted by

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

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 [12, 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

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

also been investigated, e.g. in [13]. Analogously to a zonotope, it is shown in Fig. 9 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.



(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 9: Construction of a probabilistic zonotope.

We support the following methods for probabilistic zonotopes:

- `center` – returns the center of the probabilistic zonotope.
- `display` – standard method, see Sec. 5.
- `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 [12, 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 [12, Equation 3].
- `mean` – returns the uncertain mean of a probabilistic zonotope.
- `mSigma` – converts a probabilistic zonotope to a common zonotope where for each generator, a m-sigma interval is taken.
- `mtimes` – standard method, see Sec. 5 as stated in [12, Equation 4] for numeric matrix multiplication. The multiplication of interval matrices is also supported.
- `plot` – standard method, see Sec. 5. More details can be found in Sec. 5.8.
- `plus` – standard method, see Sec. 5. Addition is realized for `probZonotope` objects with MATLAB vectors, `zonotope` objects, and `probZonotope` objects as described in [12, 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 [12, 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. 5.1.3, while the order reduction of

the probabilistic part is done by the method `probReduce`.

- `sigma` – returns the Σ matrix of a probabilistic zonotope.

5.4.1 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 enclosur 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. 10.

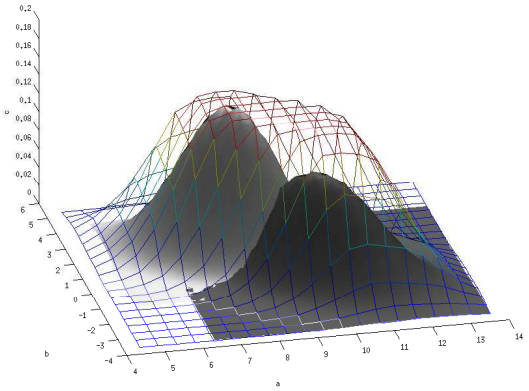


Figure 10: Sets generated in lines 10-15 of the probabilistic zonotope example in Sec. 5.4.1.

5.5 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:

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

When the intersection is unbounded, one obtains a polyhedron [14].

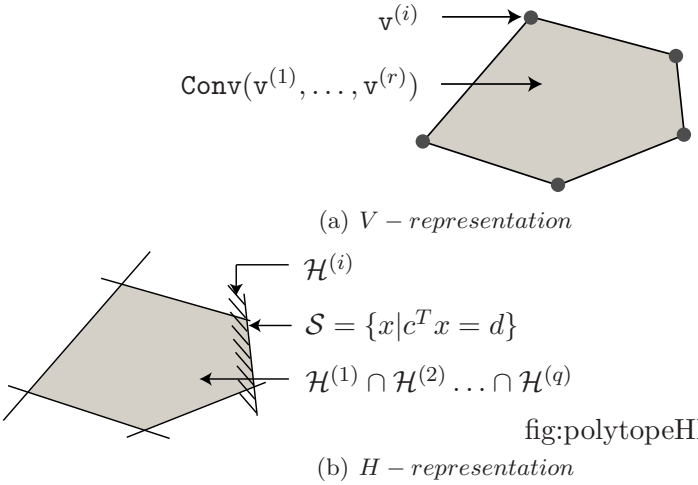
A polytope with vertex representation is defined as the convex hull of a finite set of points in the n-dimensional Euclidean space. The points are also referred to as vertices and are denoted by $\mathtt{v}^{(i)} \in \mathbb{R}^n$. A convex hull of a finite set of r points is obtained from their linear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

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

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. 11. Algorithms that convert from H- to V-representation and vice versa are presented in [15].



(a) $V-representation$

fig:polytopeHRepresentation

(b) $H-representation$

Figure 11: 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`.
- `display` – standard method, see Sec. 5.
- `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`.
- `is_empty` – returns 1 if a `mptPolytope` is empty and 0 otherwise.

- `mldivide` – computes the set difference of two `mptPolytopes`.
- `mptPolytope` – constructor of the class.
- `mtimes` – standard method, see Sec. 5 for numeric and interval matrix multiplication.
- `plot` – standard method, see Sec. 5. More details can be found in Sec. 5.8.
- `plus` – standard method, see Sec. 5 for numeric vectors and `mptPolytope` objects.
- `vertices` – returns a `vertices` object including all vertices of the polytope.
- `volume` – computes the volume of a polytope.

5.5.1 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. 12.

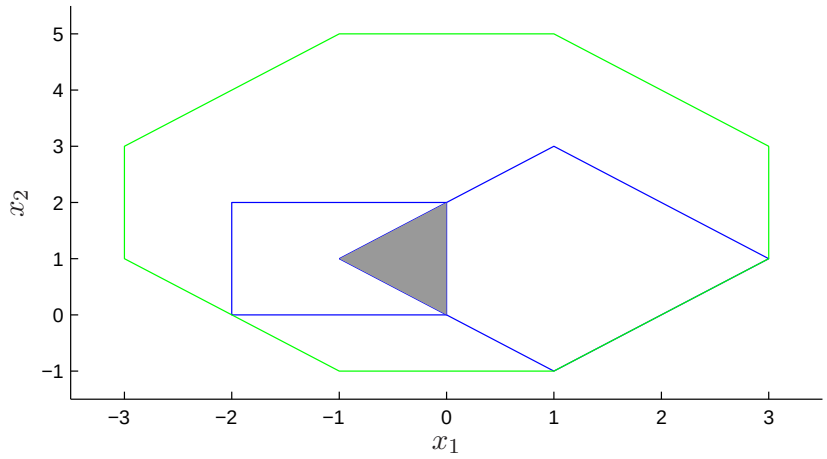


Figure 12: Sets generated in lines 13-16 of the MPT polytope example in Sec. 5.5.1.

5.6 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 [1]. 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 [1, Eq. (10)].
- `acos` – arccos(·) function as defined in [1, Eq. (6)].
- `acosh` – arccosh(·) function as defined in [1, Eq. (8)].
- `and` – computes the intersection of two `intervals` as defined in [1, Eq. (1)].
- `asin` – arcsin(·) function as defined in [1, Eq. (6)].
- `asinh` – arcsinh(·) function as defined in [1, Eq. (8)].
- `atan` – arctan(·) function as defined in [1, Eq. (6)].
- `atanh` – arctanh(·) function as defined in [1, Eq. (8)].
- `cos` – cos(·) function as defined in [1, Eq. (13)].
- `cosh` – cosh(·) function as defined in [1, 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 [1, Eq. (4)].
- `le` – overloads `<=` operator: Is one interval equal or the subset of another interval?

- `log` – natural logarithm function as defined in [1, Eq. (5)].
- `lt` – overloads `<` operator: Is one interval equal or the subset of another interval?
- `minus` – overloaded '-' operator, see [1, Eq. (2)].
- `mpower` – overloaded ^ operator (power), see [1, Eq. (9)].
- `mrdivide` – overloaded '/' operator (division), see [1, Eq. (3)].
- `mtimes` – overloaded '*' operator (multiplication), see [1, Eq. (2)] for scalars and [1, Eq. (16)] for matrices.
- `plus` – overloaded '+' operator (addition), see [1, Eq. (2)] for scalars and [1, Eq. (17)] for matrices.
- `power` – overloaded '.^' operator for intervals (power), see [1, Eq. (9)].
- `rdivide` – overloads the './' operator: provides elementwise division of two matrices.
- `sin` – $\sin(\cdot)$ function as defined in [1, Eq. (12)].
- `sinh` – $\sinh(\cdot)$ function as defined in [1, Eq. (7)].
- `sqrt` – $\sqrt{(\cdot)}$ function as defined in [1, Eq. (5)].
- `tan` – $\tan(\cdot)$ function as defined in [1, Eq. (14)].
- `tanh` – $\tanh(\cdot)$ function as defined in [1, 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

- `display` – standard method, see Sec. 5.
- `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.
- `infimum` – returns the infimum of an interval.
- `interval` – constructor of the class.
- `isempty` – returns 1 if a `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. 5. More details can be found in Sec. 5.8.
- `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.

5.6.1 Interval Example

The following MATLAB code demonstrates some of the introduced methods:

```
I1 = interval([0; -1], [3; 1]); % create interval I1
I2 = interval([-1; -1.5], [1; -0.5]); % create interval I2
Z1 = zonotope([1 1 1; 1 -1 1]); % create zonotope Z1

r = rad(I1) % obtain and display radius of I1
is_intersecting = isIntersecting(I1, Z1) % Z1 intersecting I1?
I3 = I1 & I2; % computes the intersection of I1 and I2
c = mid(I3) % returns and displays the center of I3

figure; hold on
plot(I1); % plot I1
plot(I2); % plot I2
plot(Z1,[1 2],'g'); % plot Z1
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. 13.

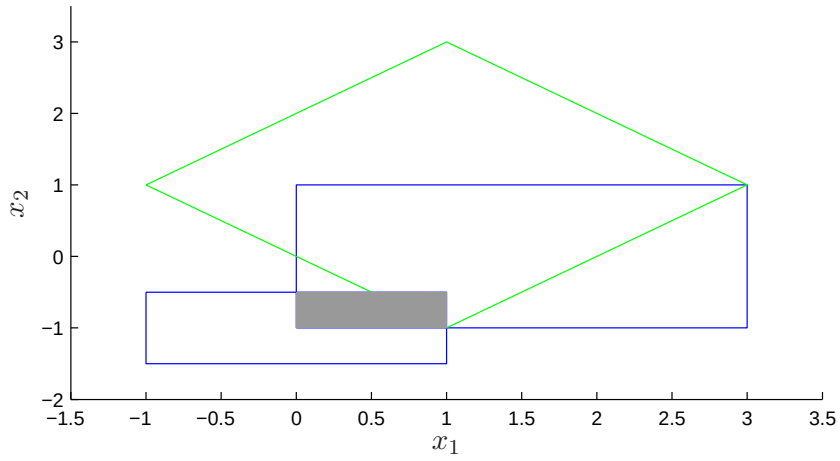


Figure 13: Sets generated in lines 11-14 of the interval example in Sec. 5.6.1.

5.7 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. 5.
- `interval` – encloses all vertices by an `interval`.
- `mtimes` – standard method, see Sec. 5 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. 5. More details can be found in Sec. 5.8.
- `plus` – standard method, see Sec. 5. Addition is only realized for `vertices` objects with MATLAB vectors.
- `vertices` – constructor of the class.
- `zonotope` – computes a zonotope that encloses all vertices according to [16, Section 3].

5.7.1 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. 14.

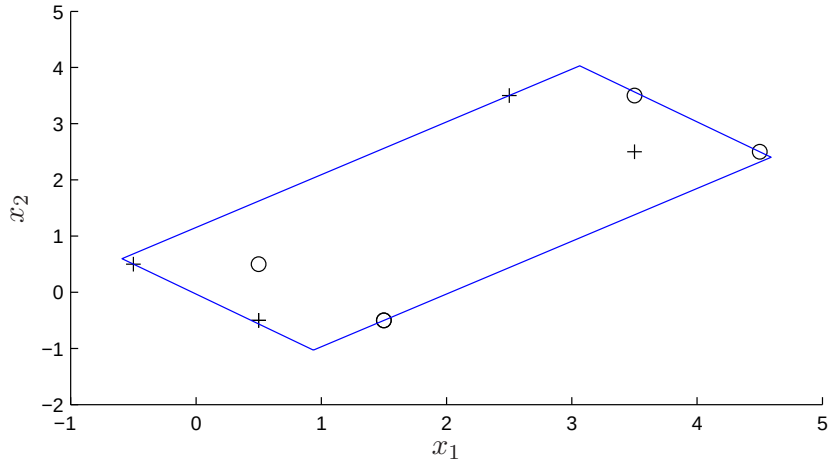


Figure 14: Sets generated in lines 11-14 of the vertices example in Sec. 5.7.1.

5.8 Plotting of Sets

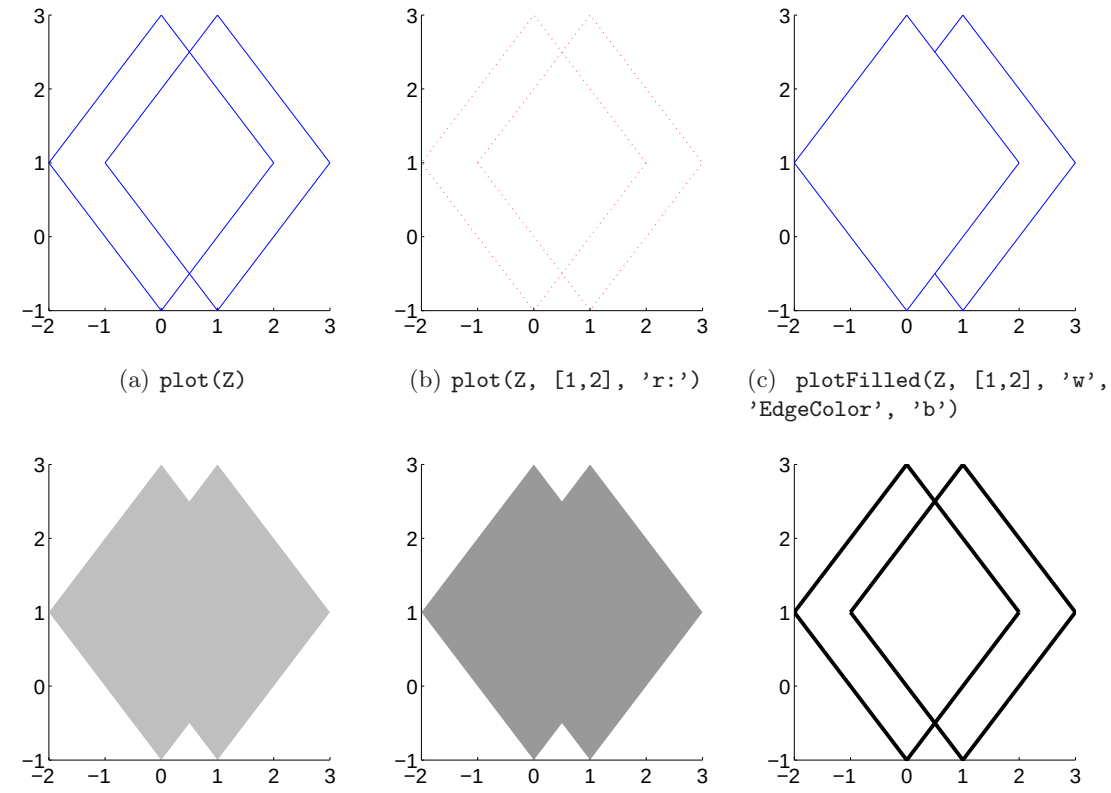
Plotting of reachable sets is performed by first projecting the set onto two dimensions. Those dimensions can be two states for plots in state space, or a state and a time interval for plots involving a time axis. A selection of plot types is presented in Fig. 15 for two zonotopes using the standard MATLAB `LineSpec`, `ColorSpec`, and `Patch` settings. The command `plot` only plots the edge, while `plotFilled` also fills the sets. The corresponding standard MATLAB functions are `plot` and `fill`, respectively.

6 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. 7.2. CORA supports the following matrix set representations:

- Matrix polytope (Sec. 6.1)
- Matrix zonotope (Sec. 6.2); specialization of a matrix polytope.
- Interval matrix (Sec. 6.3); 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



(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 15: Selection of different plot styles.

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.

6.1 Matrix Polytopes

A matrix polytope is analogously defined as a V-polytope (see Sec. 5.5):

$$\mathcal{A}_{[p]} = \Big\{ \sum_{i=1}^{r} \alpha_i V^{(i)} \Big| V^{(i)} \in \mathbb{R}^{n \times n}, \alpha_i \in \mathbb{R}, \alpha_i \geq 0, \sum_i \alpha_i = 1 \Big\}. \tag{3}$$

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. 5.5).

We support the following methods for polytope matrices:

- `display` – standard method, see Sec. 6.
- `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. 6.
- `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. 6 for numeric matrix multiplication or multiplication with another matrix polytope.
- `plot` – plots 2-dimensional projection of a matrix polytope.
- `powers` – computes the powers of a matrix zonotope up to a certain order.
- `plus` – standard method, see Sec. 6. Addition is realized 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. 6.

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 (3).

6.1.1 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
```

```
matP1 = matPolytope(P1); % instantiate matrix polytope P1

P2{1} = [-1 2; 2 -1]; % 1st vertex of matrix polytope P2
P2{2} = [-1 1; 1 -1]; % 2nd vertex of matrix polytope P2
matP2 = matPolytope(P2); % instantiate matrix polytope P2

matP3 = matP1 + matP2 % perform Minkowski addition and display result
matP4 = matP1 * matP2 % compute multiplication of and display result

intP = intervalMatrix(matP1) % compute interval matrix and display result
```

This produces the workspace output

```
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
```

6.2 Matrix Zonotopes

A matrix zonotope is defined analogously to zonotopes (see Sec. 5.1):

$$\mathcal{A}_{[z]} = \left\{ G^{(0)} + \sum_{i=1}^{\kappa} p_i G^{(i)} \middle| p_i \in [-1,1], G^{(i)} \in \mathbb{R}^{n \times n} \right\} \tag{4}$$

and is written in short form as $\mathcal{A}_{[z]} = (G^{(0)}, G^{(1)}, \ldots, G^{(\kappa)})$, where the first matrix is referred to as the *matrix center* and the other matrices as *matrix generators*. The order of a matrix zonotope is defined as $\rho = \kappa/n$. When exchanging the matrix generators by vector generators $g^{(i)} \in \mathbb{R}^n$, one obtains a zonotope (see e.g. [4]).

We support the following methods for zonotope matrices:

- `concatenate` – concatenates the center and all generators of two matrix zonotopes.
- `display` – standard method, see Sec. 6.
- `dependentTerms` – considers dependency in the computation of Taylor terms for the matrix zonotope exponential according to [8, Proposition 4.3].
- `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 [8, Section 4.4.4].
- `expmOneParam` – operator for the exponential matrix of a matrix zonotope when only one parameter is uncertain as described in [17, Theorem 1].
- `expmVertex` – computes the exponential matrix for a selected number of dominant vertices obtained by the `dominantVertices` method.

- `infNorm` – returns the maximum of the infinity norm of a matrix zonotope.
- `infNormRed` – returns a faster over-approximation of the maximum of the infinity norm of a matrix zonotope by reducing its representation size in an over-approximative way.
- `intervalMatrix` – standard method, see Sec. 6.
- `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. 6 for numeric matrix multiplication or a multiplication with another matrix zonotope according to [8, Equation 4.10].
- `plot` – plots 2-dimensional projection of a matrix zonotope.
- `plus` – standard method (see Sec. 6) for a matrix zonotope or a numerical matrix.
- `powers` – computes the powers of a matrix zonotope up to a certain order.
- `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. 6.
- `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 (4).
- `generator` – cell array of matrices $G^{(i)}$ according to (4).

6.2.1 Matrix Zonotope Example

The following MATLAB code demonstrates some of the introduced methods:

```
1  Zcenter = [1 2; 3 4]; % center of matrix zonotope Z1
2  Zdelta{1} = [1 0; 1 1]; % generators of matrix zonotope Z1
3  matZ1 = matZonotope(Zcenter, Zdelta); % instantiate matrix zonotope Z1
4
5  Zcenter = [-1 2; 2 -1]; % center of matrix zonotope Z2
6  Zdelta{1} = [0 0.5; 0.5 0]; % generators of matrix zonotope Z2
7  matZ2 = matZonotope(Zcenter, Zdelta); % instantiate matrix zonotope Z2
8
9  matZ3 = matZ1 + matZ2 % perform Minkowski addition and display result
10 matZ4 = matZ1 * matZ2 % compute multiplication of and display result
11
```

```
12 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
```

6.3 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.
- `display` – standard method, see Sec. 6.
- `dependentTerms` – considers dependency in the computation of Taylor terms for the interval matrix exponential according to [8, Proposition 4.4].
- `dominantVertices` – computes the dominant vertices of an interval matrix zonotope according to a heuristics.
- `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 [8, 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 [8, Proposition 4.1].
- `infNorm` – returns the maximum of the infinity norm of an interval matrix.
- `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. 6 for numeric matrix multiplication or a multiplication with another interval matrix according to [8, Equation 4.11].
- `plot` – plots 2-dimensional projection of an interval matrix.
- `plus` – standard method, see Sec. 6. 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.
- `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.
- `uniformSampling` – creates samples uniformly within an interval matrix.
- `vertices` – standard method, see Sec. 6.
- `volume` – computes the volume of an interval matrix by computing the volume of the corresponding interval.

6.3.1 Interval Matrix Example

The following MATLAB code demonstrates some of the introduced methods:

```
1  Mcenter = [1 2; 3 4]; % center of interval matrix M1
2  Mdelta = [1 0; 1 1]; % delta of interval matrix M1
3  intM1 = intervalMatrix(Mcenter, Mdelta); % instantiate interval matrix M1
4
5  Mcenter = [-1 2; 2 -1]; % center of interval matrix M2
6  Mdelta = [0 0.5; 0.5 0]; % delta of interval matrix M2
7  intM2 = intervalMatrix(Mcenter, Mdelta); % instantiate interval matrix M2
8
9  intM3 = intM1 + intM2 % perform Minkowski addition and display result
10 intM4 = intM1 * intM2 % compute multiplication of and display result
11
12 matZ = matZonotope(intM1) % compute matrix zonotope and display result
```

This produces the workspace output

```
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

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

7 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. 7.1)
- Linear systems with uncertain fixed parameters (Sec. 7.2)
- Linear systems with uncertain varying parameters (Sec. 7.3)
- Linear probabilistic systems (Sec. 7.4)
- Nonlinear systems (Sec. 7.5)
- Nonlinear systems with uncertain fixed parameters (Sec. 7.6)
- Nonlinear differential-algebraic systems (Sec. 7.7)

Each class for continuous dynamics inherits from the parent class `contDynamics`. This class itself is inherited from the `handle` class. 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, a dedicated method has to be implemented. Since for reachability analysis, multiple instances of the same dynamics are not required, the instantiation from a `handle` 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:

- `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 simulation of the system. It can be set of 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.

In addition, each class realizes at least 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, therefor the methods are not listed for the individual classes. We mainly focus on the method `initReach`, which is computed differently for each class.

7.1 Linear Systems

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

$$\dot{x}(t) = Ax(t) + Bu(t), \quad x(0) \in \mathcal{X}_O \subset \mathbb{R}^n, \quad u(t) \in \mathcal{U} \subset \mathbb{R}^n \tag{5}$$

For the computation of reachable sets, we use the equivalent system

$$\dot{x}(t) = Ax(t) + \tilde{u}(t), \quad x(0) \in \mathcal{X}_O \subset \mathbb{R}^n, \quad \tilde{u}(t) \in \tilde{\mathcal{U}} = B \otimes \mathcal{U} \subset \mathbb{R}^n, \tag{6}$$

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

7.1.1 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 (6). 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 [5, Section 3.2], the reachable set for the first time interval $\tau_0 = [0, r]$ is computed as shown in Fig. 16:

1. Starting from $\mathcal{X}_O$, 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 $\mathcal{X}_O$ 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 $\tilde{\mathcal{U}}_\Delta$.

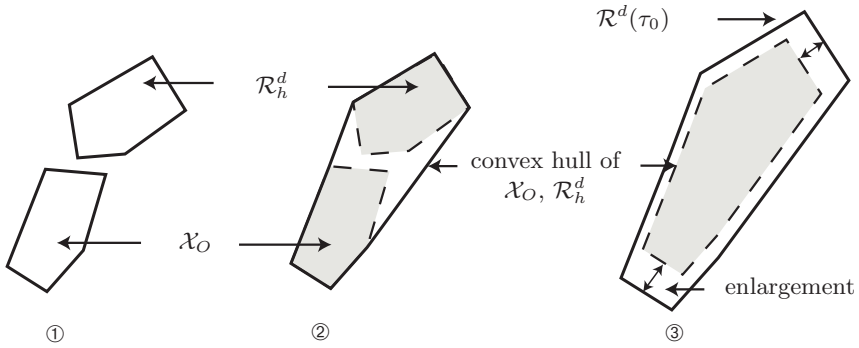


Figure 16: Steps for the computation of an over-approximation of the reachable set for a linear system.

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 [18, Proposition 2]. A more conservative approach previously used [5, Equation 3.2,3.3].
- `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 [18, Section 4]. A more conservative approach previously used [5, Proposition 3.1], which can only be used in combination with [5, 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 [5, Theorem 3.1]. As noted in [18, Theorem 2], it is required that the input set is convex. The error term in [18, Theorem 2] is slightly better, but is computationally more expensive so that the algorithm form [5, Theorem 3.1] is used.

7.2 Linear Systems with Uncertain Fixed Parameters

This class extends linear systems by uncertain parameters that are fixed over time:

$$\dot{x}(t) = A(p)x(t) + \tilde{u}(t), \quad x(0) \in \mathcal{X}_O \subset \mathbb{R}^n, \quad p \in \mathcal{P}, \quad \tilde{u}(t) \in \tilde{\mathcal{U}} = \{B(p) \otimes \mathcal{U} | \mathcal{U} \subset \mathbb{R}^n, p \in \mathcal{P}\}, \tag{7}$$

The set of state and input matrices is denoted by

$$\mathcal{A} = \{A(p) | p \in \mathcal{P}\}, \quad \mathcal{B} = \{B(p) | p \in \mathcal{P}\} \tag{8}$$

An alternative is to define each parameter as a state variable $\tilde{x}_i$ with the trivial dynamics $\dot{\tilde{x}}_i = 0$. The result is a nonlinear system that can be handled as described in Sec. 7.5. The problem of which approach to use for any particular case is still open.

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 (8). The set of matrices can be represented by any matrix set introduced in Sec. 6.
- `B` – set of input matrices $\mathcal{B}$, see (8). The set of matrices can be represented by any matrix set introduced in Sec. 6.
- `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 [5, Theorem 3.2].
- `mappingMatrixSet` – set of exponential matrices, see Sec. 6.
- `E` – remainder of matrix exponential computation.
- `F` – uncertain matrix to bound the error for time interval solutions, see e.g. [5, Proposition 3.1].
- `inputF` – uncertain matrix to bound the error for time interval solutions of inputs, see e.g. [5, Proposition 3.4].
- `inputCorr` – additional uncertainty of the input solution if origin is not contained in input set, see [5, Equation 3.9].
- `Rinput` – reachable set of the input solution, see Sec. 7.1.
- `Rtrans` – reachable set of the input u_c, see Sec. 7.1.
- `RV` – reachable set of the input $\tilde{\mathcal{U}}_\Delta$, see Sec. 7.1.
- `sampleMatrix` – possible matrix $\hat{A}$ such that $\hat{A} \in \mathcal{A}$.

7.2.1 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 fixed 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

$$\mathcal{R}_h^d = e^{\mathcal{A}r}\mathcal{X}_O = \{e^{Ar}x_0 | A \in \mathcal{A}, x_0 \in \mathcal{X}_O\}. \tag{9}$$

Similarly, the reachable set due to the input solution changes as described in [5, 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 [8, 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 [8, 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 [8, Theorem 1].

7.3 Linear Systems with Uncertain Varying Parameters

This class extends linear systems with uncertain, but fixed parameters to linear systems with time-varying parameters:

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

The set of state matrices can be represented by any matrix set introduced in Sec. 6. The provided methods of the class are identical to the ones in Sec. 7.2, except that the computation is based on [18].

Since the `linVarSys` 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 (8). The set of matrices can be represented by any matrix set introduced in Sec. 6.
- `B` – set of input matrices $\mathcal{B}$, see (8). The set of matrices can be represented by any matrix set introduced in Sec. 6.
- `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 [5, Theorem 3.2].
- `mappingMatrixSet` – set of exponential matrices, see Sec. 6.
- `power` – tba
- `E` – remainder of matrix exponential computation.
- `F` – uncertain matrix to bound the error for time interval solutions, see e.g. [5, Proposition 3.1].
- `inputF` – uncertain matrix to bound the error for time interval solutions of inputs, see e.g. [5, Proposition 3.4].
- `inputCorr` – additional uncertainty of the input solution if origin is not contained in input set, see [5, Equation 3.9].
- `Rinput` – reachable set of the input solution, see Sec. 7.1.
- `Rtrans` – reachable set of the input u_c, see Sec. 7.1.
- `sampleMatrix` – possible matrix $\hat{A}$ such that $\hat{A} \in \mathcal{A}$.

7.4 Linear Probabilistic Systems

In contrast to all other systems, we consider stochastic properties in the class `linProbSys`. The system under consideration is defined by the following linear stochastic differential equation (SDE) which is also known as the multivariate Ornstein-Uhlenbeck process [19]:

$$\begin{aligned} \dot{x} &= Ax(t) + u(t) + C\xi(t), \\ x(0) \in \mathbb{R}^n,\, u(t) &\in \mathcal{U} \subset \mathbb{R}^n,\, \xi \in \mathbb{R}^m \end{aligned} \tag{10}$$

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 (10) 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 (10) $\forall t \in [0, r]$, $u(t) \in \mathcal{U}$, $f_{\mathbf{X}}(x, 0) = f_0\}$. The enclosing probabilistic hull for a time interval is defined as $\bar{f}_{\mathbf{X}}(x, [0, r]) = \sup\{\bar{f}_{\mathbf{X}}(x, t) | t \in [0, r]\}$.

7.4.1 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. 7.1.1. 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. 7.1.
- `tie` (**t**ime **i**nterval **e**rror – computes the `tie` similarly as for linear systems in Sec. 7.1.
- `inputSolution` – computes the reachable set due to the input according to the superposition principle of linear systems. The computation is performed as suggested in [12, Secttion VI.B].

7.5 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 [5, Section 3.4] and by polynomial systems as presented in [10]. 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 $\mathcal{X}_O \subset \mathbb{R}^n$ 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:

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

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.

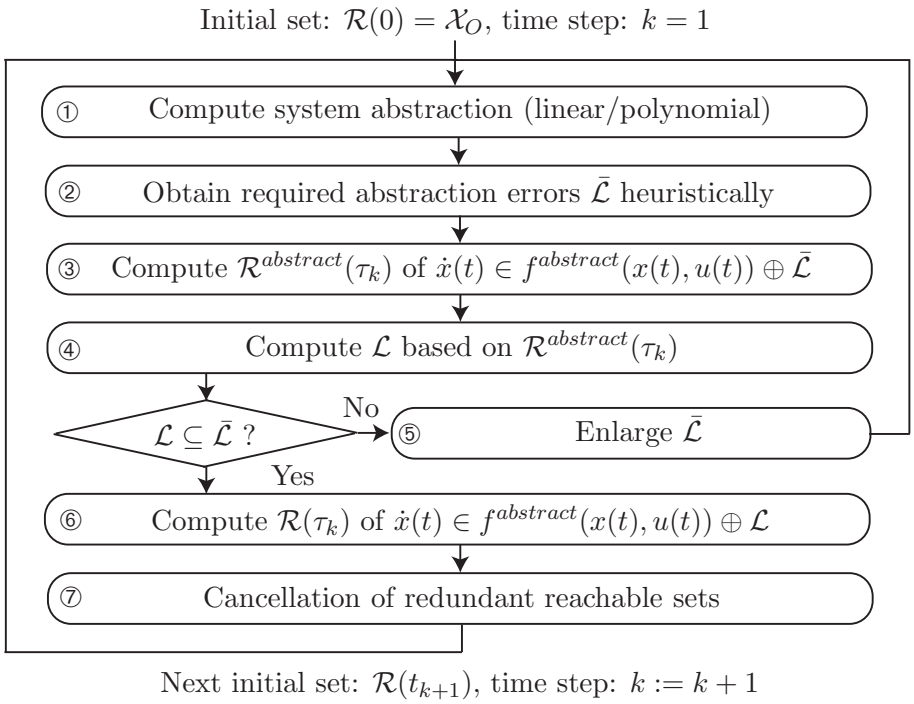


Figure 17: Computation of reachable sets – overview.

A brief visualization of the overall concept for computing the reachable set is shown in Fig. 17. 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 (6) or after introducing $z = [x^T,\, u^T]^T$ a polynomial system of the form

$$\begin{aligned} \dot{x}_i = f^{abstract}(x, u) =& w_i + \frac{1}{1!}\sum_{j=1}^{o} C_{ij} z_j(t) + \frac{1}{2!}\sum_{j=1}^{o}\sum_{k=1}^{o} D_{ijk} z_j(t) z_k(t) \\ &+ \frac{1}{3!}\sum_{j=1}^{o}\sum_{k=1}^{o}\sum_{l=1}^{o} E_{ijkl} z_j(t) z_k(t) z_l(t) + \ldots \end{aligned} \tag{11}$$

The set of abstraction errors $\mathcal{L}$ ensures that $f(x, u) \in f^{abstract}(x, u) \oplus \mathcal{L}$, which allows the reachable set to be computed in an over-approximative way.

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

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

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

⑤ 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 nexessity of splitting reachable sets is indicated in the workspace outputs using the keyword `split` and 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.

7.5.1 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 private functions take care of the required 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. The distinction between the reachable set computation by polynomial abstractions and linear abstractions is made by the computation of the reachable set due to the abstraction error:
  - `errorSolutionQuad` – for polynomial abstraction.
  - `errorSolution` – for linear abstraction.
- `linearize` – linearizes the nonlinear system.
- `linError_mixed_noInt` – computes the linearization error without use of interval arithmetic according to [6, Theorem 1].
- `linError_thirdOrder` – computes linearization errors according to [10, Section 4.1].
- `linError` – easiest, but also most conservative computation of the linearization error according to [20, Proposition 1].

7.6 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.

7.7 Nonlinear Differential-Algebraic Systems

The class `nonlinDASys` considers time-invariant, semi-explicit, index-1 DAEs without parametric uncertainties since they are not yet implemented. The extension to parametric uncertainties can

be done using the methods applied in Sec. 7.6. Using the vectors of differential variables x, algebraic variables y, and inputs u, the semi-explicit DAE can generally be written as

$$\begin{aligned} \dot{x} &= f(x(t), y(t), u(t)) \\ 0 &= g(x(t), y(t), u(t)), \\ [x^T(0), y^T(0)]^T &\in \mathcal{R}(0), \quad u(t) \in \mathcal{U}, \end{aligned} \tag{12}$$

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 [21, 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 [22, 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 [21, 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 [6, 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 [6, Proposition 2].

Since the `nonlinDASys` class is written using the new structure for object oriented programming in MATLAB, it 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.
- `hessianAbs` – tba
- `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.

8 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[5]. 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. 12.2.1), where the velocity of the ball changes instantaneously when hitting the ground.

The formal definition of the hybrid automaton is similarly defined as in [16]. 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[6] $\texttt{inv}$: $\mathcal{V} \to 2^{\mathcal{X}}$, 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: $\dot{x} = \texttt{f}(v, x, u, p)$.

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

$$x' = K_{(v_i,v_j)}\, x + l_{(v_i,v_j)}, \tag{13}$$

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.

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

[6] $2^{\mathcal{X}}$ is the power set of $\mathcal{X}$.

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. 18.

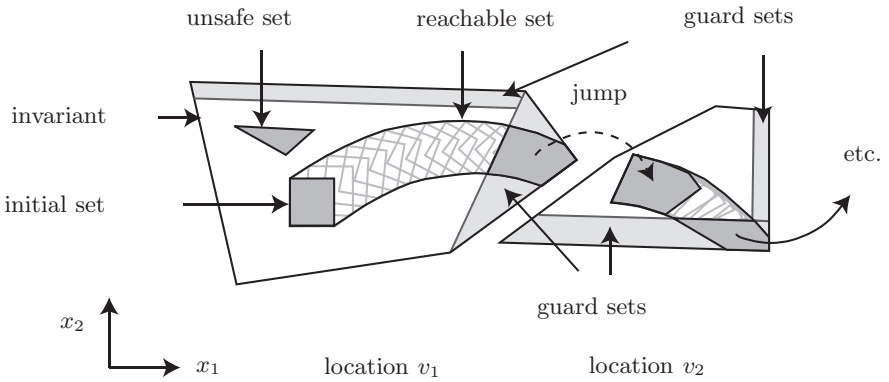


Figure 18: Illustration of the reachable set of a hybrid automaton.

8.1 Simulation of Hybrid Automata

While the reachable set computation of hybrid systems as performed in CORA is described in several publications, see e.g. [5,23,24], 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:

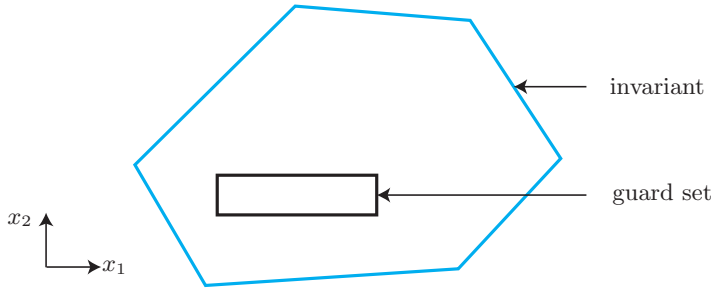
① Preparation 1: Guard sets and invariants can be specified by any set representation that CORA offers. For simulation purposes, all set representations are transformed into a halfspace representation as illustrated in Fig. **??**. This is performed by transforming intervals, zonotopes, and zonotope bundles to a polytope, see Fig. 2. Next, of all polytopes the halfspace generation is obtained. Guard 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. **??**.

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.

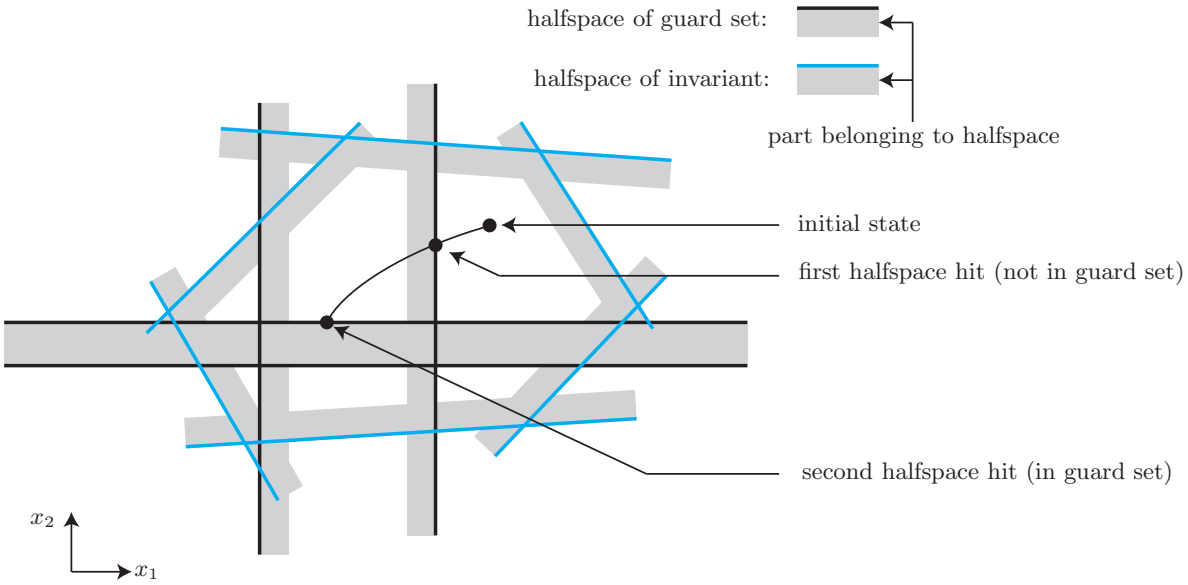
CORA automatically generates an event function for each halfspace, where the simulation is stopped when the halfspace of the invariant is left (`direction=+1`) and stopped for halfspaces of guard sets when the halfspace is entered (`direction=-1`). In any case, the simulation will stop.

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. 19(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. 20.

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.



(a) Considered location.



(b) Simulation using halfspaces.

Figure 19: Illustration of the algorithm for simulating a hybrid automaton.

8.2 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.

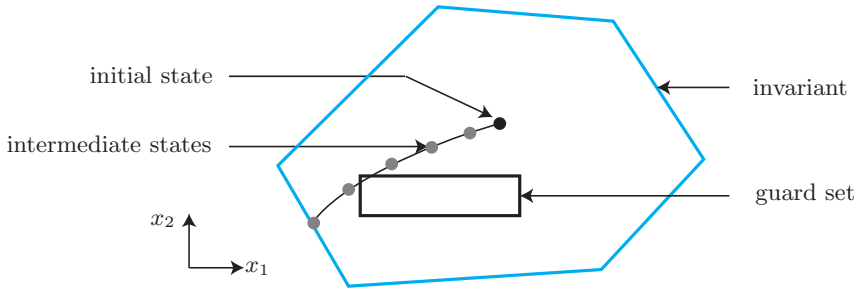


Figure 20: Guard intersections can be missed when one only checks whether interndiate states are in any guard set.

8.3 Location

Each location consists of:

- `invariant` – specified by a set representation of Sec. 5.
- `transitions` – cell array of objects of the class `transition`.
- `contDynamics` – specified by a continuous dynamics of Sec. 7.
- `name` – saved as a string describing the location.
- `id` – unique number of the location.

We mainly support the following methods for locations:

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

8.4 Transition

Each transition consists of

- `guard` – specified by a set representation of Sec. 5.
- `reset` – struct containing the information for a linear reset.
- `target` – `id` of the target location when the transition occurs.
- `inputLabel` – input event to communicate over events.
- `outputLabel` – output event to communicate over events.

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').

9 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:

- `allSegmentIntervalHulls` – generates all interval hulls of the partitioned space.
- `cellCandidates` – finds possible cells that might intersect with a continuous set over-approximated by its bounding box (interval hull); more cell indices are returned than actually intersect.
- `cellCenter` – return center of specified cell.
- `cellIndices` – returns cell indices given a set of cell coordinates.
- `cellIntersection2` – returns the volumes of a polytope P intersected with touched cells C_i.
- `cellSegment` – returns cell coordinates given a set of cell indices.
- `display` – displays the parameters of the partition in the MATLAB workspace.
- `findSegment` – find segment index for given state space coordinates.
- `findSegments` – return segment indices intersecting with a given interval hull.
- `nrOfStates` – returns the number of discrete states of the partition.
- `partition` – constructor of the class.
- `segmentIntervals` – returns intervals of segment.
- `segmentPolytope` – returns polytope of segment.
- `segmentZonotope` – returns zonotope of segment.

10 Options for Reachability Analysis

Most parameters for the computation of reachable sets are controlled by a `struct` called `options`. These are the most important fields:

- `tStart` – start time of the analysis.
- `tFinal` – final time of the analysis.
- `x0` – initial state.
- `R0` – initial set of states.
- `u` – constant input for simulations.
- `uTrans` – u_c: transition of the uncertain input set $\tilde{\mathcal{U}}_\Delta$.

- `uTransVec` – varying u_c for each time step: transition of the uncertain input set $\tilde{\mathcal{U}}_\Delta$.
- `U` – uncertain input set $\tilde{\mathcal{U}}_\Delta$.
- `originContained` – flag whether the origin is contained in the set of uncertain inputs $\tilde{\mathcal{U}}$ (1: yes, 0: no).
- `timeStep` – step size $t_{k+1} - t_k$.
- `taylorTerms` – considered Taylor terms for the exponential matrix.
- `zonotopeOrder` – maximum order of zonotopes.
- `intermediateOrder` – order up to which no interval methods are used in matrix set computations.
- `advancedLinErrorComp` – flag to enable advanced linearization error computation (1: on, 0: off).
- `tensorOrder` – maximum order up to which tensors are considered in the abstraction of the system.
- `errorOrder` – maximum zonotope order for the computation of nonlinear maps.
- `maxError` – maximum allowed abstraction errors before a reachable set is split.
- `reductionInterval` – number of time steps after which redundant reachable sets are removed.

11 Unit Tests

To better ensure that all functions in CORA work as they should, despite changes in the code, CORA contains a number of unit tests. Those unit test 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_`.

12 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.

12.1 Continuous Dynamics

12.1.1 Linear Dynamics

For linear dynamics, a simple academic example from [5, 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; %input for simulation
48 options.U=0.5*zonotope([zeros(5,1),diag([0.2, 0.5, 0.2, 0.5, 0.5])]); %input set
49
50 options.path = [coraroot '/contDynamics/stateSpaceModels'];
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*[1 1 1]);
    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 $t_f = 5$.

12.1.2 Linear Dynamics with Uncertain Parameters

For linear dynamics with uncertain parameters, we use the transmission line example from [8, 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 [8, 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.

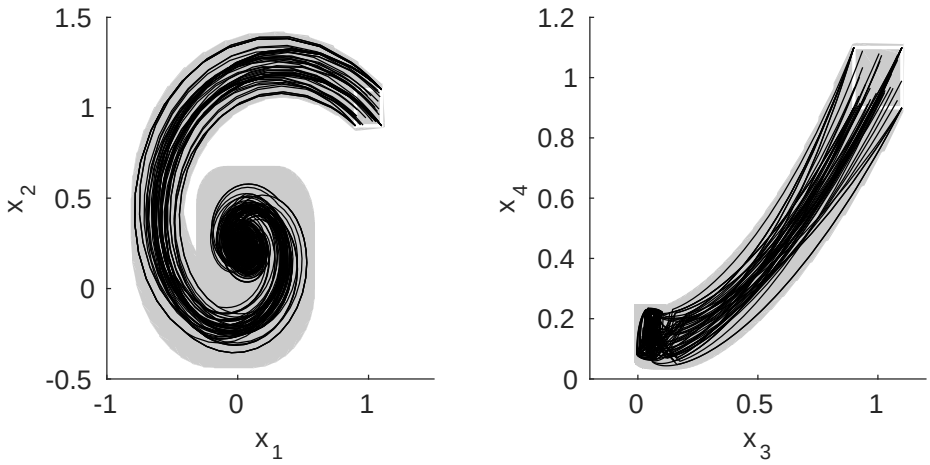


Figure 21: Illustration of the reachable set of the linear example. The white box shows the initial set and the black lines show simulated trajectories.

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 of reachable set using matrix zonotopes: ',num2str(tComp)]);

tic
Rcont2 = reach(linSys2, options);
tComp = toc;
disp(['computation time of reachable set 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*[1 1 1]);
166 end
167 
168 %------------------------------------------------------------------------
```

The reachable set and the simulation are plotted in Fig. 22 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. 23.

12.1.3 Nonlinear Dynamics

For nonlinear dynamics, several examples are presented.

Tank System The first example is the tank system from [20] where water flows from one tank into another one. This example can be used to study the effect of water power plants on the

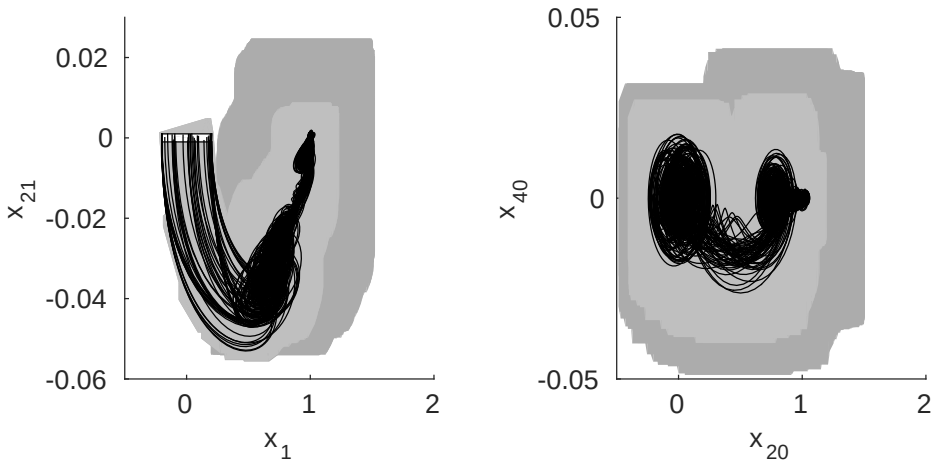


Figure 22: Illustration of the reachable set of the transmission example. The light gray shows the reachable set using matrix zonotopes and the dark gray shows the results using interval matrices. A white box shows the initial set and the black lines are simulated trajectories.

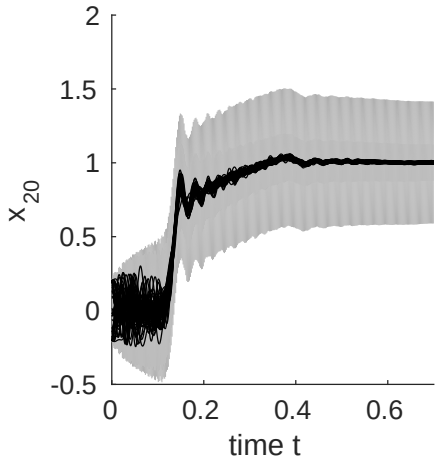


Figure 23: Illustration of the reachable set of the transmission example over time. The light gray shows the reachable set using matrix zonotopes and the dark gray shows the results using interval matrices. Black lines show simulated trajectories.

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 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_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  options.path = [coraroot '/contDynamics/stateSpaceModels'];
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 systems 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. 28 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 [20] to demonstrate that one can obtain a solution even if the linearization error becomes too large by splitting the reachable set. Later, in [10] an improved method is presented that requires less splitting. This example demonstrates the capabilities of the simpler approach presented in [20]. Due to the similarity of the MATLAB code compared to the previous tank example, we only present the reachable set in Fig. 25.

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*

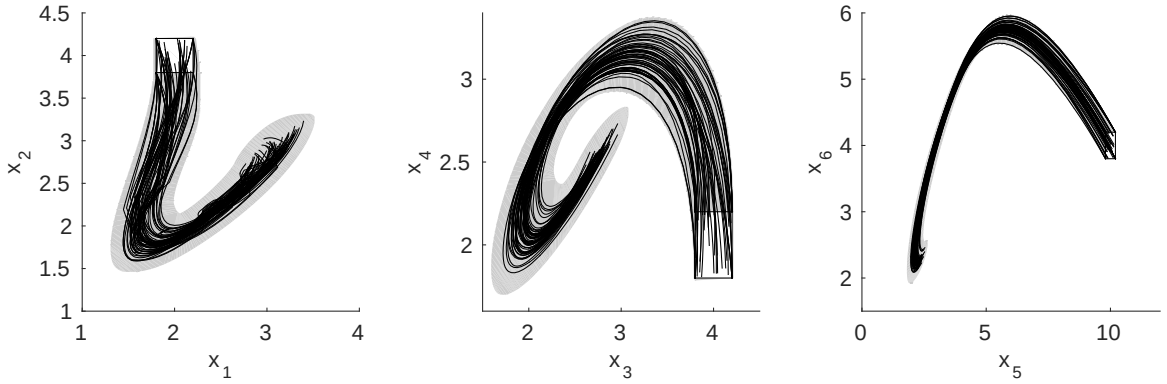


Figure 24: Illustration of the reachable set of the linear example. The white box shows the initial set and the black lines show simulated trajectories.

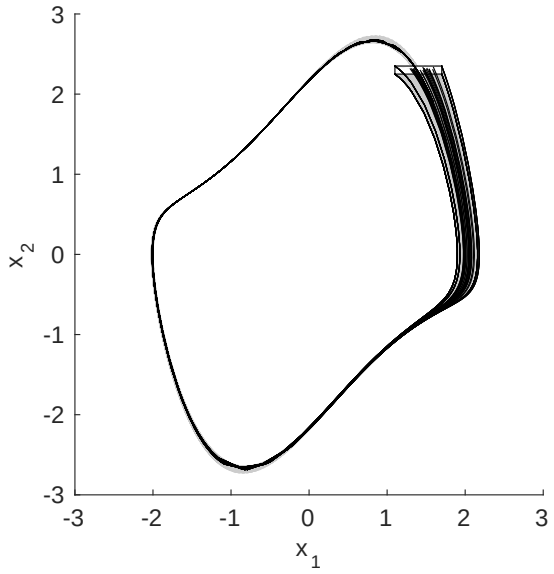


Figure 25: Illustration of the reachable set of the Van der Pol oscillator. The white box shows the initial set and the black lines show simulated trajectories.

is used as presented in [10]. 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. 26.

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. [17,25,26]. In CORA, this only requires to change 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. 27, where the reference trajectory is plotted in red.

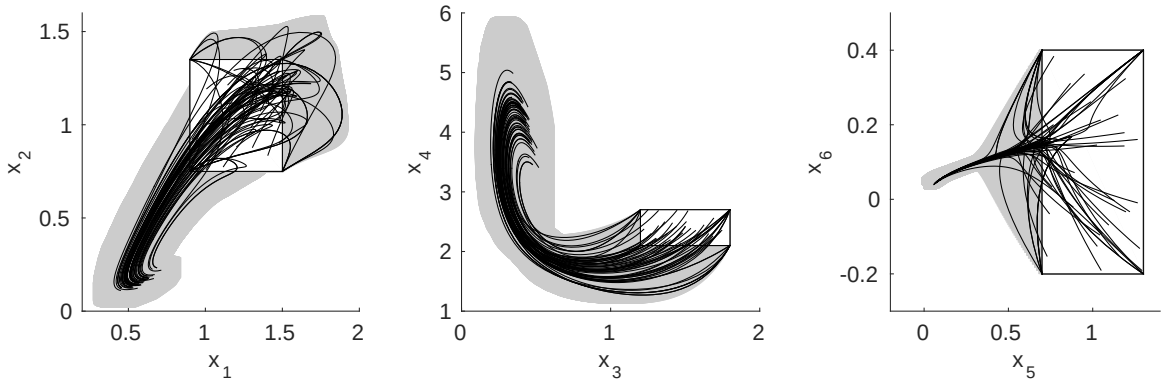


Figure 26: Illustration of the reachable set of the seven-dimensional example for non-convex set representation. The white box shows the initial set and the black lines show simulated trajectories.

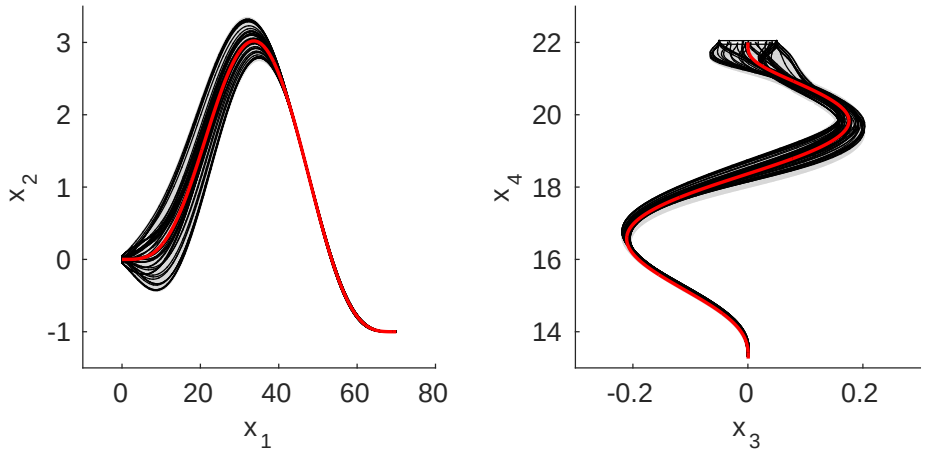


Figure 27: Illustration of the reachable set of the seven-dimensional example for non-convex set representation. The white box shows the initial set and the black lines show simulated trajectories.

12.1.4 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 use again the tank system of which we now the reachable set 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; %has to be zero for nonlinear systems!!

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. 28 for a time horizon of $t_f = 0.7$.

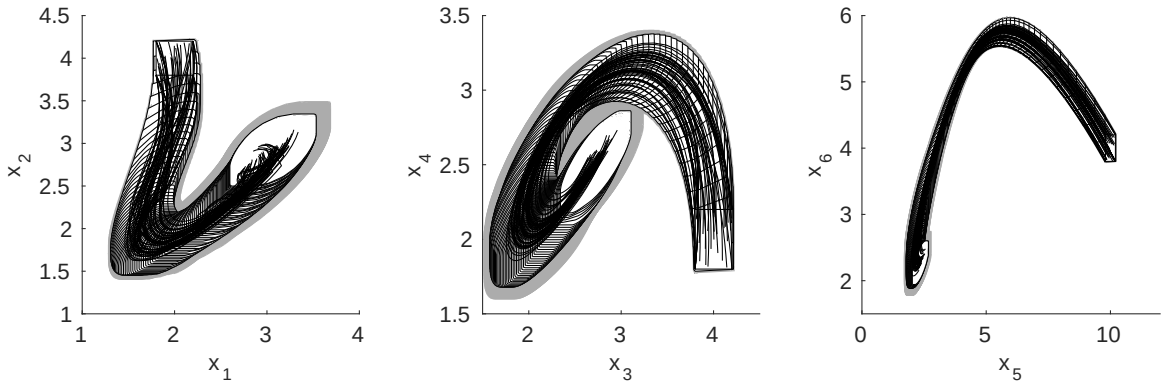


Figure 28: Illustration of the reachable set of the linear example. The gray region shows the reachable set with uncertain parameters, while the white area shows the reachable set without uncertain parameters. Another white box shows the initial set and the black lines show simulated trajectories.

12.1.5 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 [6, 27, 28].

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.path = [coraroot '/contDynamics/stateSpaceModels'];
29 options.tensorOrder = 1;
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. 29 for a time horizon of $t_f = 5$.

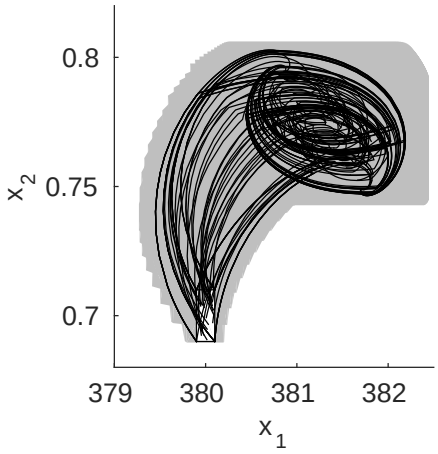


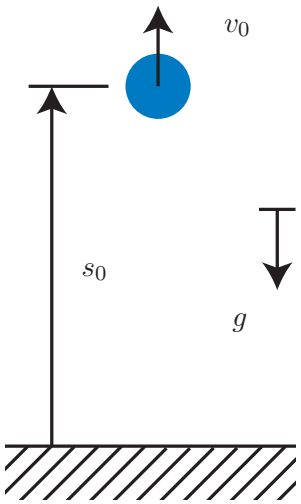
Figure 29: Illustration of the reachable set of nonlinear differential-algebraic example. The white box shows the initial set and the black lines show simulated trajectories.

12.2 Hybrid Dynamics

As already described in Sec. 8, CORA can compute reachable sets of mixed discrete/continuous or so-called hybrid systems. The difficulty in computing reachable set 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. [23,29–34]), but the mapping-based approach has shown great scalability for some examples [24]. Determining advantages and disadvantages of both methods require further research.

12.2.1 Bouncing Ball Example

We demonstrate the syntax of CORA for the well-known bouncing ball example, see e.g. [35, Section 2.2.3]. Given is a ball in Fig. 30 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. 8 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 30: 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. 32 for a time horizon of $t_f = 5$.

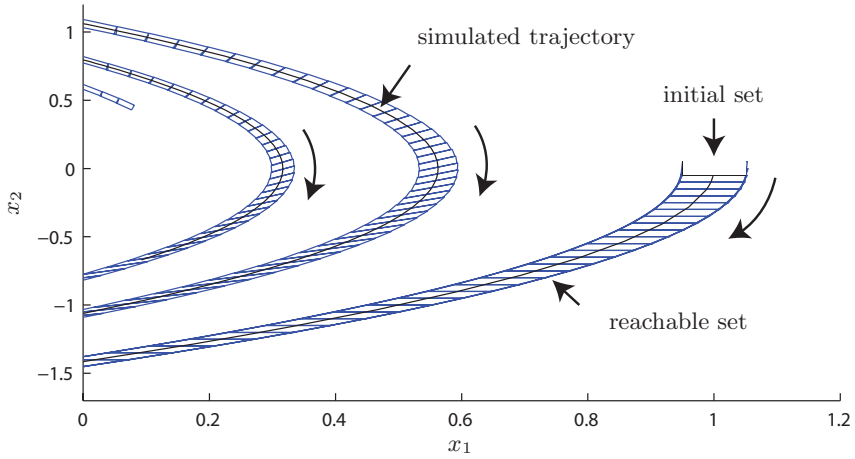


Figure 31: Illustration of the reachable set of the bouncing ball. The black box shows the initial set and the black line shows the simulated trajectory.

12.2.2 Powertrain Example

The powertrain example is taken out of [24, 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 length, we are not presenting it in the manual, but 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. 32 for a time horizon of $t_f = 2$.

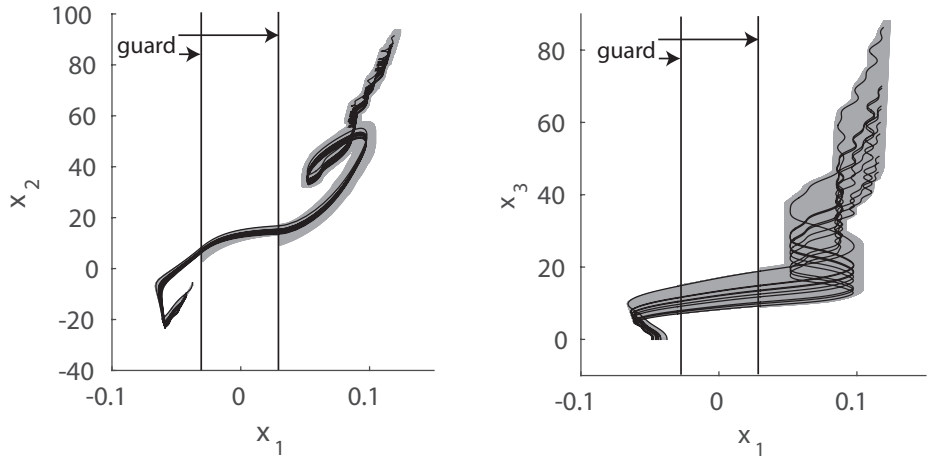


Figure 32: Illustration of the reachable set of the bouncing ball. The black box shows the initial set and the black line shows the simulated trajectory.

13 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 [28]. The efficiency of the algorithms used means it is even possible to verify problems online, i.e. while they are in operation [26].

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.

Acknowledgment

The author gratefully acknowledges financial support by the European Commission project UnCoVerCPS under grant number 643921.

A Migrating the `intervalhull` Class into the `interval` Class

This table should help automatically renaming old functions to make own code compatible with CORA 2016. Details on the functionality of each method can be found in Sec. 5.6.

old command	**new command**
`abs`	[-1,1]`supremum(abs)`
`and`	`and`
`bloat`	`enlarge`
`center`	`mid`
`display`	`display`
`edgeLength`	`2*rad`
`enclose`	`hull`
`eventFcn`	–
`get`	–
`gridPoints`	`gridPoints`
`halfspace`	–
`in`	`isIntersecting`
`infimum`	`infimum`
`isempty`	`isempty`
`le`	`le`
`lt`	`lt`
`mptPolytope`	–
`mtimes`	`mtimes`
`or`	`hull`
`plot`	`plot`
`plus`	`plus`
`polytope`	`polytope`
`radius`	`enclosingRadius`
`rdivide`	`rdivide`
`relativeGridPoints`	–
`subsref`	`subsref` (different semantics)
`supremum`	`supremum`
`vertices`	`vertices`
`volume`	`volume`
`zonotope`	`zonotope`

B Licensing

CORA is released under the GPLv3.

C 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 are addressing a limited audience.

If you have questions or suggestions, please contact us through `http://www6.in.tum.de/`.

D Contributions

The following people have contributed so far (alphabetical order of last name):

- Matthias Althoff
- Victor Charlent
- Dmitry Grebenyuk
- Daniel Heß

References

[1] 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.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

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

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

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

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

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