CORA 2015 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.

1 Philosophy and Architecture

The **CO**ntinuous **R**eachability **A**nalyzer (CORA)[1] is a MATLAB toolbox for prototype 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 [1] and (b) over-approximative computations qualify for formal verification. Thus, CORA computes over-approximations for particular points in time $\mathcal{R}(t) \supseteq \mathcal{R}^e(t)$ and for time intervals: $\mathcal{R}([t_0, t_f]) = \bigcup_{t \in [t_0, t_f]} \mathcal{R}(t)$.

CORA is built with the aim to construct one's own reachable set computation in a 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

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

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 toolboxes MPT[2] and INTLAB[3], which are also written in MATLAB. Results of CORA can be easily transferred to those toolboxes and vice versa. Please note that not all functionalities have yet been ported to version 3 of the MPT so that MPT2 should be used. The next CORA release will work with MPT3.

Since this text 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 [2].

2 Installation

The software does not require any installation, except that the path for CORA has to be set in MATLAB. Besides CORA, the following toolboxes have to be downloaded and included in the MATLAB path:

- Multi-Parametric Toolbox 2[4] (MPT toolbox), which is designed for parametric optimization, computational geometry and model predictive control. CORA only uses the computational geometry capabilities for polytopes. The MPT toolbox is free software provided by ETH Zürich.
- INTLAB[5] is a MATLAB toolbox for inteval and affine arithmatic. Interval arithmatic is promarily used in CORA to obtain over-approximate bounds for the abstraction error computation, e.g. from nonlinear to linear systems. The tool requires to run `startintlab` for an initial installation. It is developed by the Hamburg University of Technology and used to be a free tool. In the long run, we try to avoid the connection with INTLAB since it is not free anymore. If you use an older, free version, you might experience problems with the latest MATLAB versions. Strangely, we could fix some issues by rearranging the order of the include path in MATLAB.

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

2.1 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[6]: Classes for set representations (Sec. 3), classes for matrix set representations (Sec. 4), classes for the analysis of continuous dynamics (Sec. 5), classes for the analysis of hybrid dynamics (Sec. 6), and a class for the partitioning of the state space (Sec. 7).

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

[2] `http://control.ee.ethz.ch/~mpt/2/`
[3] `http://www.ti3.tu-harburg.de/rump/intlab/`
[4] `http://control.ee.ethz.ch/~mpt/2/`
[5] `http://www.ti3.tu-harburg.de/rump/intlab/`
[6] `http://www.uml.org/`

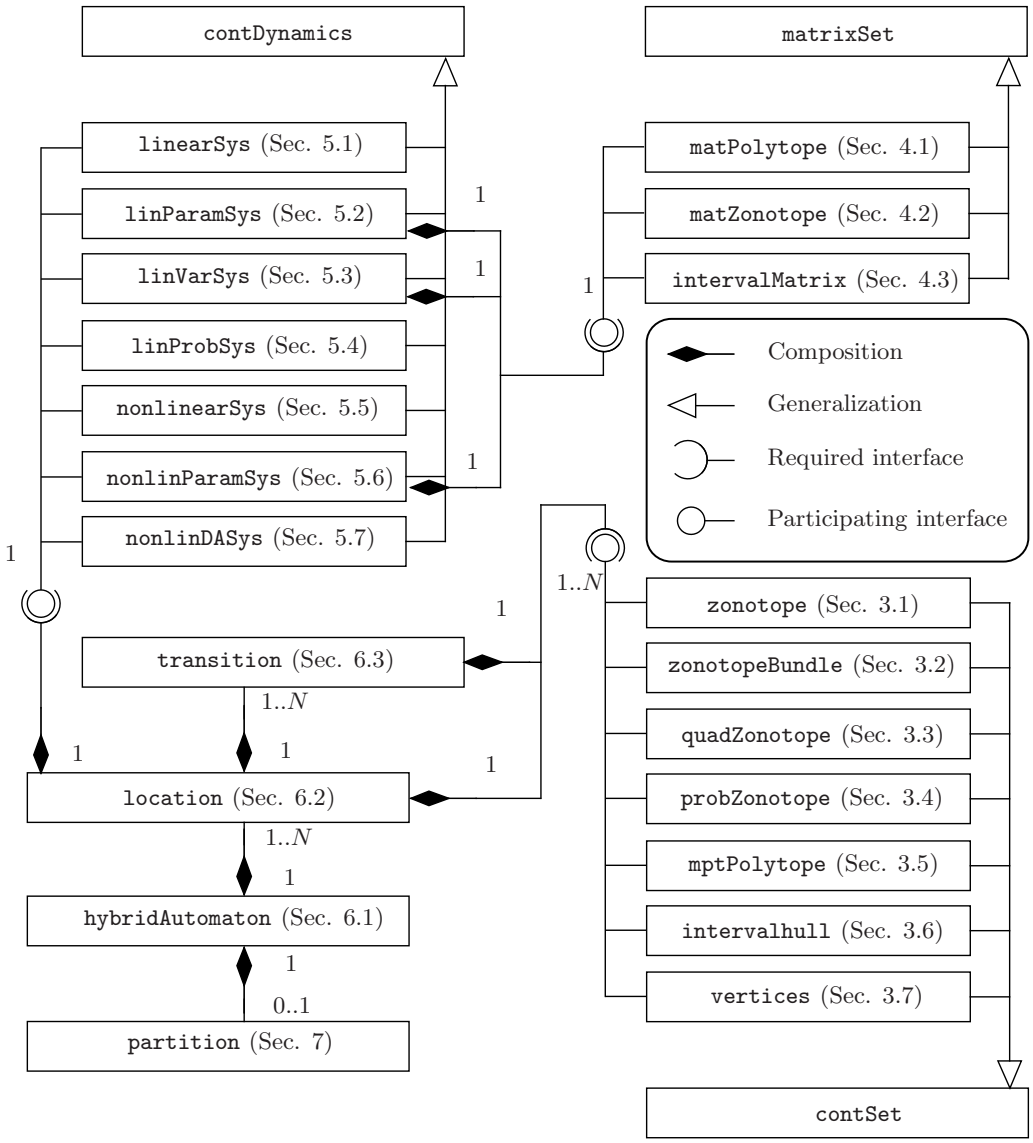


Figure 1: Unified Modeling Language (UML) class diagram of CORA 2015.

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. 6.

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 for those systems very efficient algorithms exist that are not applicable to more general systems.

3 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.

Important operations for sets are:

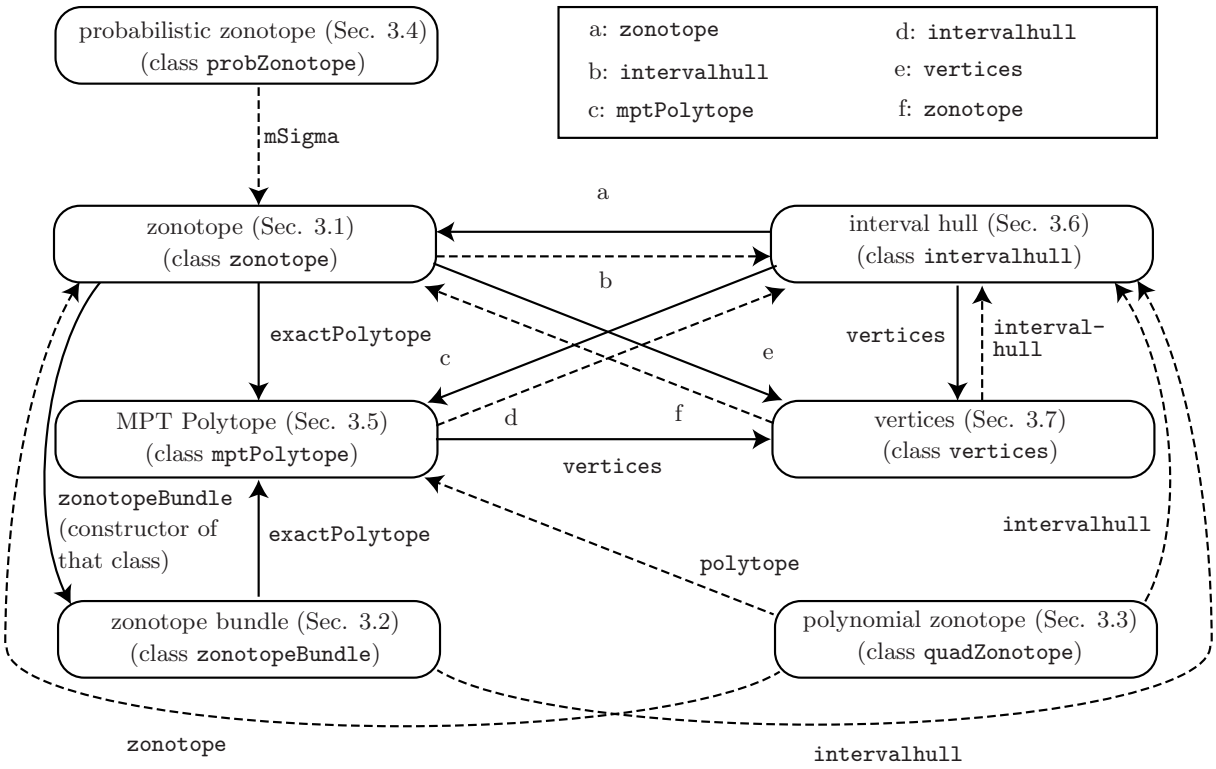


Figure 2: Set conversions supported. Solid arrows represent exact conversions, while dashed arrows represent over-approximative conversions. The arrows are labeled by the corresponding method to carry out the conversion.

- `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 Minkowsi sum $\{x + y|x \in Z1, y \in Z2\}$.
- `intervalhull`: Returns an interval hull that encloses the set (see Sec. 3.6).

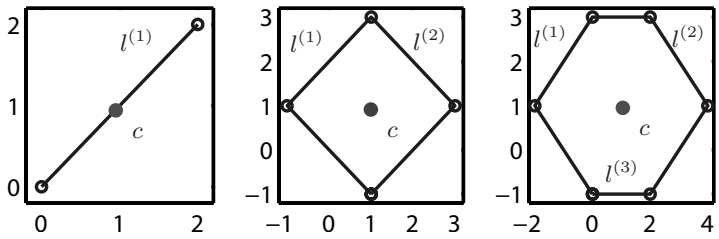
3.1 Zonotopes

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 [3].



(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:

- `cartesianProduct` – returns the Cartesian product of two zonotopes.
- `center` – returns the center of the zonotope.
- `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. 3.
- `enclose` – generates a zonotope that encloses two zonotopes of equal dimension according to [4, Equation 2.2 + subsequent extension].
- `enlarge` – enlarges the generators of a zonotope by a vector of factors for each dimension.
- `exactPolytope` – returns an exact polytope in halfspace representation according to [4, Theorem 2.1].
- `inParallelotope` – checks if a zonotope is a subset of a parallelotope, where the latter is represented as a zonotope.
- `intervalhull` – standard method, see Sec. 3. More details can be found in [4, Proposition 2.2].
- `mtimes` – standard method, see Sec. 3. More details can be found in Sec. 3.1.1.
- `plot` – standard method, see Sec. 3. More details can be found in Sec. 3.8.
- `plus` – standard method, see Sec. 3. More details can be found in Sec. 3.1.2.
- `polytope` – returns an over-approximating polytope in halfspace representation according to heuristics as proposed in [4, Sec. 2.5.6.].
- `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 $i = 1 \ldots n$, `quadraticMultiplication` computes $\{\varphi | \varphi_i = x^T Q^{(i)} x,\, x \in \mathcal{Z}\}$ as described in [5, Lemma 1].
- `randPoint` – generates a random point within a zonotope.
- `reduce` – returns an over-approximating zonotope with fewer generators as detailed in Sec. 3.1.3.

- `split` – splits a zonotope into two or more zonotopes that enclose the original zonotope. More details can be found in Sec. 3.1.4.
- `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 [6, p.40].
- `zonotope` – constructor of the class.

3.1.1 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	-	-
`intval`	INTLAB class	[7]
`intervalMatrix`	Sec. 4.3	[4, Theorem 3.3]
`matZonotope`	Sec. 4.2	[8, Sec. 4.4.1]

3.1.2 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. 3.1	[4, Equation 2.1]

3.1.3 Method `reduce`

The zonotope reduction returns an over-approximating zonotope with less generators as described in [4, Proposition 2.5]. Table 3 lists some of the implemented reduction techniques. The standard reduction technique is `'girard'`.

3.1.4 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

Table 3: Reduction techniques for zonotopes.

reduction technique	primary use	literature
`girard`	Reduction of high to medium order	[3, Sec. 3.4]
`MethA`	Reduction to parallelotope	Method A in [4, Sec. 2.5.5]
`MethB`	Reduction to parallelotope	Method B in [4, Sec. 2.5.5]
`MethC`	Reduction to parallelotope	Method C in [4, Sec. 2.5.5]

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 **NEEDS RENAMING!**	[4, Proposition 3.8]
`directionSplit`	splits all generators in one direction	—
`directionSplitBundle`	exact split using zonotope bundles	[9, Section V.A]

3.1.5 Zonotope Example

The following MATLAB code demonstrates some of the introduced methods:

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

Z3 = Z1 + Z2; % Minkowski addition
Z4 = A*Z3; % linear map

figure
plot(Z1,[1 2],'b'); % plot Z1 in blue
plot(Z2,[1 2],'g'); % plot Z2 in green
plot(Z3,[1 2],'r'); % plot Z3 in red
plot(Z4,[1 2],'k'); % plot Z4 in black

P = exactPolytope(Z4) % convert to and display halfspace representation
IH = intervalhull(Z4) % convert to and display intervah hull

figure
plot(Z4); % plot Z4
plot(IH,[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.

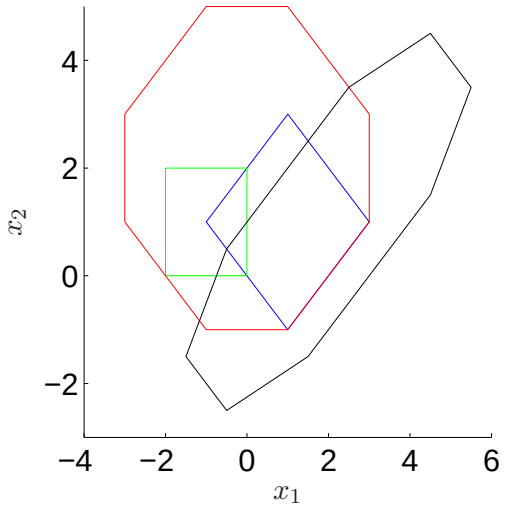


Figure 4: Zonotopes generated in lines 9-12 of the zonotope example in Sec. 3.1.5.

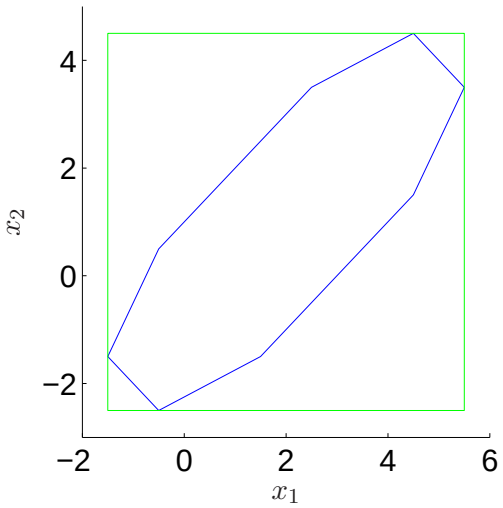


Figure 5: Sets generated in lines 18-19 of the zonotope example in Sec. 3.1.5.

3.2 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.
- `display` – standard method, see Sec. 3.
- `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.
- `exactPolytope` – returns an exact polytope in halfspace representation. Each zonotope is converted to halfspace representation according to [4, Theorem 2.1] and later all obtained H polytopes are intersected.
- `intervalhull` – standard method, see Sec. 3. More details can be found in [9, Proposition 6].
- `mtimes` – standard method, see Sec. 3. More details can be found in [9, Proposition 1].
- `plot` – standard method, see Sec. 3. More details can be found in Sec. 3.8.
- `plus` – standard method, see Sec. 3. More details can be found in [9, Proposition 2].
- `polytope` – returns an over-approximating polytope in halfspace representation. For each zonotope the method `polytope` of the class `zonotope` in Sec. 3.1 is called.
- `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. 3.1 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 `exactPolytope` and using a volume computation for polytopes.
- `zonotopeBundle` – constructor of the class.

3.2.1 Zonotope Bundle Example

The following MATLAB code demonstrates some of the introduced methods:

```
Z{1} = zonotope([1 1 1; 1 -1 1]); % create zonotope Z1;
Z{2} = zonotope([-1 1 0; 1 0 1]); % create zonotope Z2;
Zb = zonotopeBundle(Z); % instantiate zonotope bundle from Z1, Z2
vol = volume(Zb) % compute and display volume of zonotope bundle

figure
plot(Z1); % plot Z1
plot(Z2); % plot Z2
plot(Zb,[1 2],'gray'); % plot Zb in gray (filled)
```

This produces the workspace output

```
vol =

    1.0000
```

The plot generated in lines 7-9 is shown in Fig. 6.

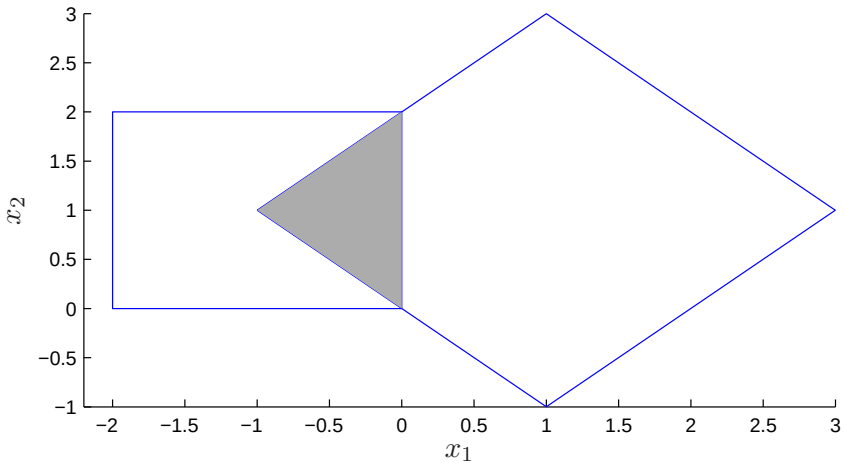


Figure 6: Sets generated in lines 7-9 of the zonotope bundle example in Sec. 3.2.1.

3.3 Polynomial Zonotopes

Zonotopes are a very efficient representation for reachability analysis of linear systems [3] and of nonlinear systems that can be well abstracted by linear differential inclusions [4]. 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

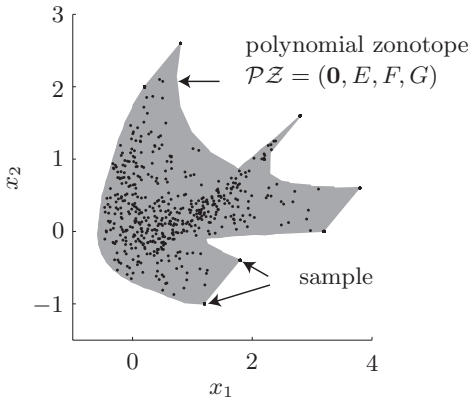
$$E^{[i]} = [\underbrace{f^{([i],1,1,\ldots,1)}}_{=:e^{([i],1)}} \ldots \underbrace{f^{([i],p,p,\ldots,p)}}_{=:e^{([i],p)}}] \text{ (all indices are the same value)},$$

$$F^{[i]} = [f^{([i],1,1,\ldots,1,2)}\ f^{([i],1,1,\ldots,1,3)} \ \ldots\ f^{([i],1,1,\ldots,1,p)}$$
$$f^{([i],1,1,\ldots,2,2)}\ f^{([i],1,1,\ldots,2,3)} \ \ldots\ f^{([i],1,1,\ldots,2,p)}$$
$$f^{([i],1,1,\ldots,3,3)} \ldots] \text{ (not all indices are the same value)},$$

$$G = [g^{(1)} \ \ldots\ g^{(q)}],$$

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.



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

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.

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

- `cartesianProduct` – returns the Cartesian product of a `quadZonotope` and a `zonotope`.
- `center` – returns the starting point c.
- `display` – standard method, see Sec. 3.
- `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.
- `generators` – returns the generators of a `quadZonotope`.
- `intervalhull` – standard method, see Sec. 3. The interval hull is obtained by over-approximating the `quadZonotope` by a `zonotope` and subsequent application of its `intervalhull` 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 `intervalhull`. 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. 3 as stated in [9, Equation 14] for numeric matrix multiplication. As described in Sec. 3.1.1 the multiplication of interval matrices is also supported, whereas the implementation for matrix zonotopes is not yet implemented.
- `plot` – standard method, see Sec. 3. More details can be found in Sec. 3.8.
- `plus` – standard method, see Sec. 3. 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)).
- `polytope` – returns an over-approximating polytope in halfspace representation by first over-approximating by a `zonotope` object and subsequently applying its `polytope` method.
- `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. 3.3.1.
- `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].

3.3.1 Method `reduce`

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

3.3.2 Polynomial Zonotope Example

The following MATLAB code demonstrates some of the introduced methods:

```
1 c = [0;0]; % starting point
2 E1 = diag([-1,0.5]); % generators of factors with identical indices
```

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	[3, Sec. 3.4]

```
3  E2 = [1 1; 0.5 0.3]; % generators of factors with identical indices
4  F = [-0.5; 1]; % generators of factors with different indices
5  G = [0.3; 0.3]; % independent generators
6
7  qZ = quadZonotope(c,E1,E2,F,G); % instantiate quadratic zonotope
8  Z = zonotope(qZ) % over-approximate by a zonotope
9
10 figure
11 plot(Z); % plot Z
12 plot(qZ,[1 2],'darkgray',7); % 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.

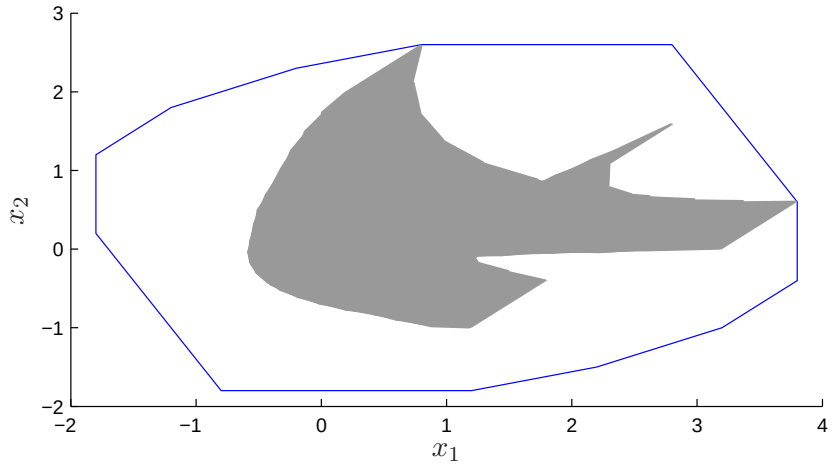


Figure 8: Sets generated in lines 11-12 of the polynomial zonotope example in Sec. 3.3.2.

3.4 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 [7]. 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:

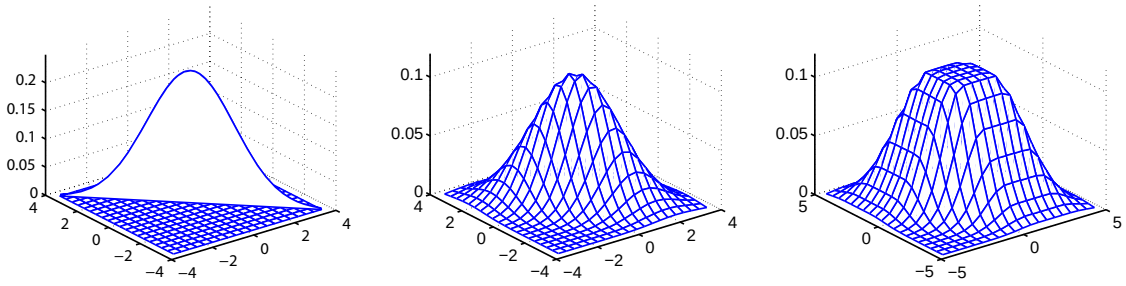
$$\mathcal{Z}_g = c + \sum_{i=1}^{q} \mathcal{N}^{(i)}(0, 1) \cdot \underline{g}^{(i)},$$

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 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 $(0, g^{(1\ldots 2)}, \underline{g}^{(1\ldots 2)})$.

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. 3.
- `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].

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

- `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. 3 as stated in [12, Equation 4] for numeric matrix multiplication. The multiplication of interval matrices is also supported.
- `plot` – standard method, see Sec. 3. More details can be found in Sec. 3.8.
- `plus` – standard method, see Sec. 3. 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. 3.1.3, while the order reduction of the probabilitsic part is done by the method `probReduce`.
- `sigma` – returns the Σ matrix of a probabilistic zonotope.

3.4.1 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.

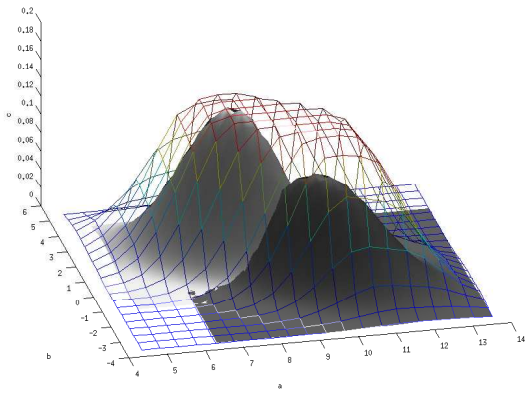


Figure 10: Sets generated in lines 10-15 of the probabilistic zonotope example in Sec. 3.4.1.

3.5 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:

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

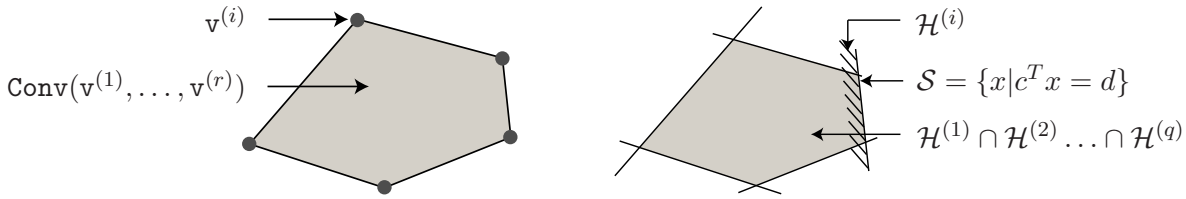
Given the convex hull operator `Conv()`, a convex and bounded polytope can be defined in vertex representation as follows:

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].

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`.



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

Figure 11: Possible representations of a polytope.

- `display` – standard method, see Sec. 3.
- `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.
- `intervalhull` – encloses a `mptPolytope` by an `intervalhull`.
- `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. 3 for numeric and interval matrix multiplication.
- `plot` – standard method, see Sec. 3. More details can be found in Sec. 3.8.
- `plus` – standard method, see Sec. 3 for numeric vectors and `mptPolytope` objects.
- `vertices` – returns a `vertices` object including all vertices of the polytope.
- `volume` – computes the volume of a polytope.

3.5.1 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 = exactPolytope(Z1); % convert zonotope Z1 to halfspace representation
P2 = exactPolytope(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
plot(P1); % plot P1
plot(P2); % plot P2
plot(P3,[1 2],'g'); % plot P3
plot(P4,[1 2],'darkgray'); % 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.

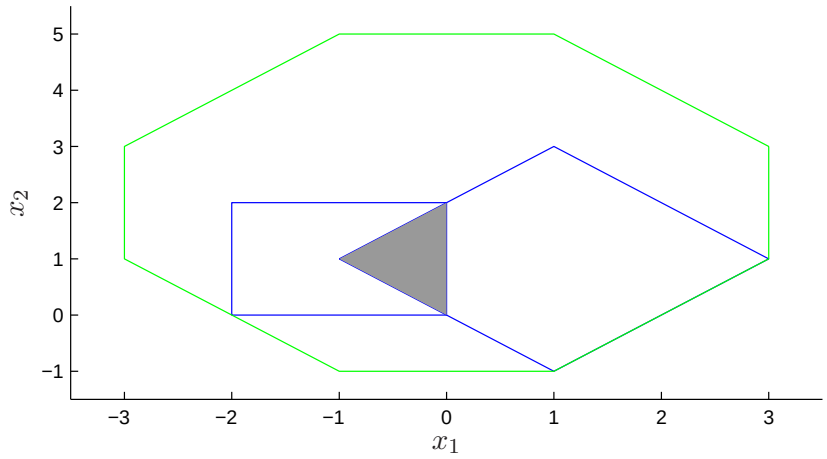


Figure 12: Sets generated in lines 13-16 of the MPT polytope example in Sec. 3.5.1.

3.6 Interval Hulls

An interval hull $\mathcal{I}$ is the closest axis-aligned box of a set. It can easily be represented as a multidimensional interval: $\mathcal{I} = [\underline{x}, \overline{x}]$, $\underline{x} \in \mathbb{R}^n$, $\overline{x} \in \mathbb{R}^n$, $\forall i : \underline{x}_i \leq \overline{x}_i$. We provide the following methods for interval hulls:

- `abs` – returns the absolute value bound of an interval hull: $|\mathcal{I}|_i = \sup\{|x_i| | x \in \mathcal{I}\}$.
- `and` – computes the intersection of two `intervalhulls`.
- `center` – returns the center of the `intervalhull`.
- `display` – standard method, see Sec. 3.

- `edgeLength` – determines the edge lengths of the interval hull.
- `enclose` – computes an interval hull that encloses two interval hulls.
- `halfspace` – generates halfspace representation of the intervalhull.
- `in` – determines if a `zonotope` is enclosed by the `intervalhull`.
- `inf` - returns the infimum of an intervalhull.
- `interval` – converts an `intervalhull` to INTLAB intervals.
- `intervalhull` – constructor of the class.
- `is_empty` – returns 1 if a `intervalhull` is empty and 0 otherwise.
- `le` – overloads `<=` operator: Is one interval hull equal or the subset of another interval hull?
- `lt` – overloads `<` operator: Is one interval hull equal or the subset of another interval hull?
- `mptPolytope` – converts an `intervalhull` object to a `mptPolytope` object.
- `mtimes` – standard method, see Sec. 3 for numeric matrix multiplication. The multiplication of interval matrices is also supported. Since interval hulls are not closed under linear mappings, the result is an over-approximation.
- `or` – computes union of interval hulls. Since interval hulls are not closed under unification, the result is an over-approximation.
- `plot` – standard method, see Sec. 3. More details can be found in Sec. 3.8.
- `plus` – standard method, see Sec. 3 for numeric vectors and `intervalhull` objects.
- `polytope` – converts an interval hull object to a polytope.
- `radius` – computes radius of an enclosing circle.
- `rdivide` – overloads the `./` operator; elementwise division of intervals by a vector.
- `sup` – returns the supremum of an intervalhull.
- `vertices` – returns a `vertices` object including all vertices.
- `volume` – computes the volume of an interval hull.
- `zonotope` – converts an `intervalhull` object to a `zonotope` object.

3.6.1 Interval Hull Example

The following MATLAB code demonstrates some of the introduced methods:

```
IH1 = intervalhull([0 3; -1 1]); % create interval hull IH1
IH2 = intervalhull([-1 1; -1.5 -0.5]); % create interval hull IH2
Z1 = zonotope([1 1 1; 1 -1 1]); % create zonotope Z1

l = edgeLength(IH1) % obtain and display edge length of IH1
is_intersecting = in(IH1, Z1) % determines and displays if Z1 is enclosed by IH1
IH3 = IH1 & IH2; % computes the intersection of IH1 and IH2
c = center(IH3) % returns and displays the center of IH3

figure
plot(IH1); % plot IH1
```

```
12 plot(IH2); % plot IH2
13 plot(Z1,[1 2],'g'); % plot Z1
14 plot(IH3,[1 2],'darkgray'); % plot IH3
```

This produces the workspace output

```
l =

     3
     2


is_intersecting =

     1


c =

    0.5000
   -0.7500
```

The plot generated in lines 11-14 is shown in Fig. 13.

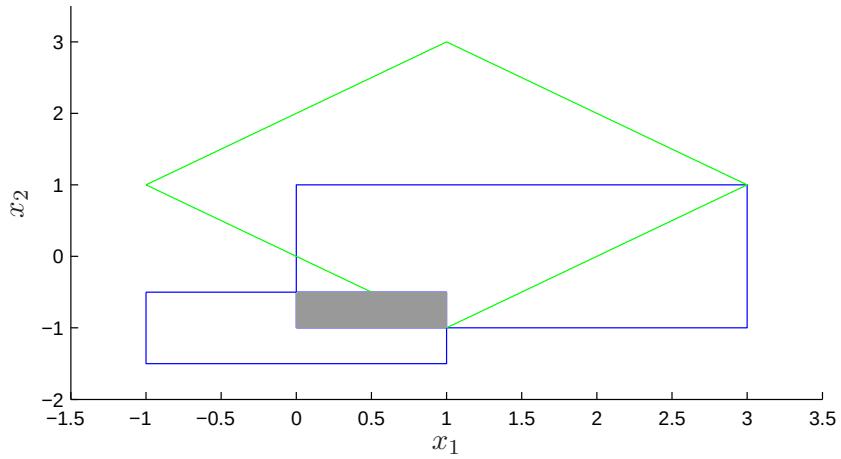


Figure 13: Sets generated in lines 11-14 of the interval hull example in Sec. 3.6.1.

3.7 Vertices

The `vertices` class has two main purposes: It is the class that performs the plotting since all other set representations are first converted to vertices to perform the plotting. Second, if one defines a point cloud as a set of potential vertices, this class computes enclosures of all points. The following methods are implemented:

- `collect` – collects cell arrays (MATLAB-specific container) of vertices.
- `display` – standard method, see Sec. 3.
- `intervalhull` – encloses all vertices by an `intervalhull`.

- `mtimes` – standard method, see Sec. 3 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. 3. More details can be found in Sec. 3.8.
- `plus` – standard method, see Sec. 3. 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].

3.7.1 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.

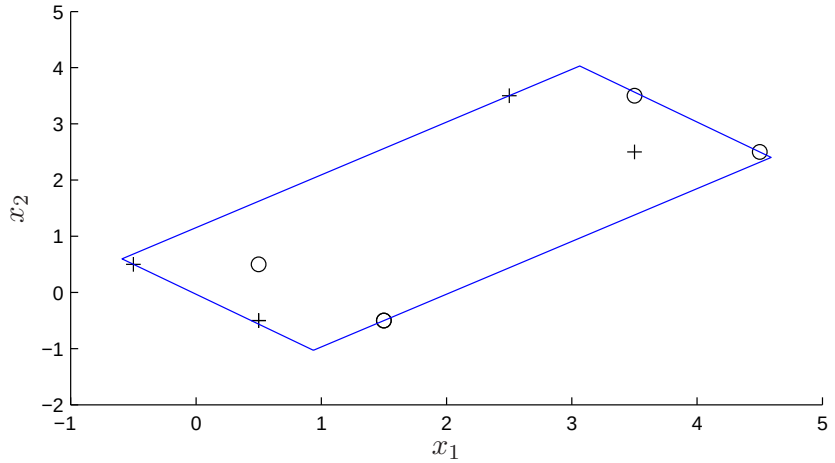
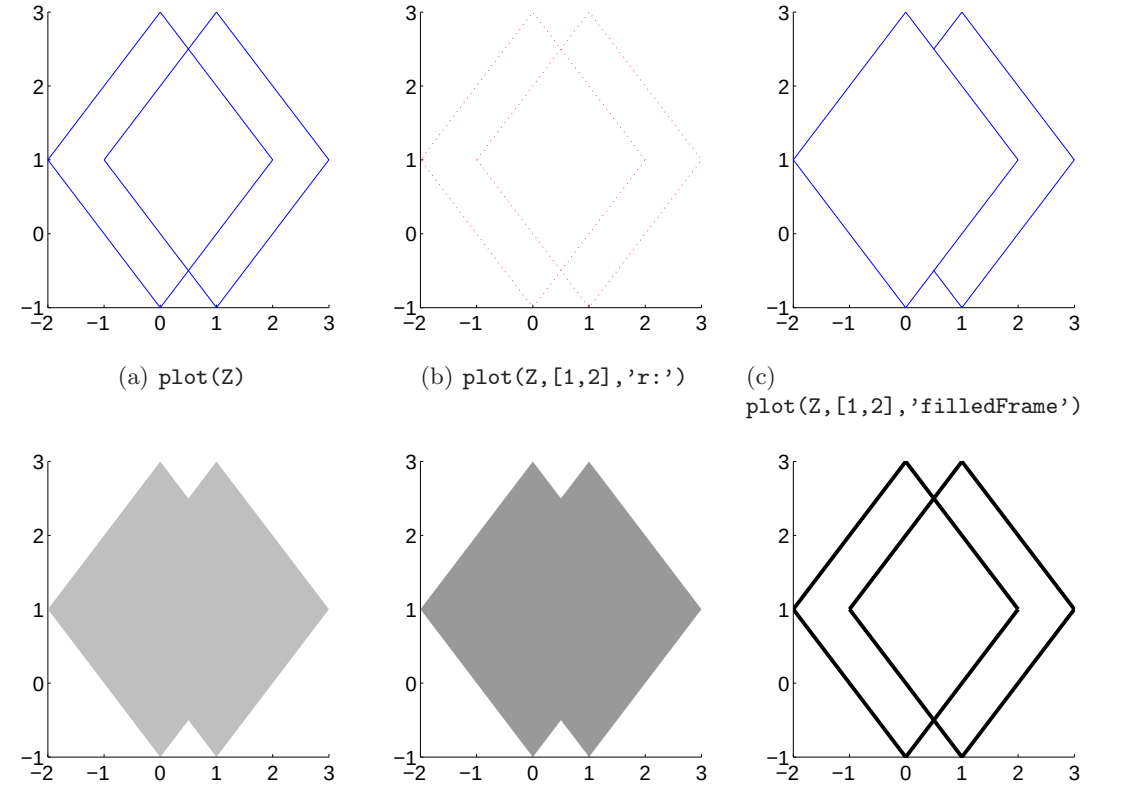


Figure 14: Sets generated in lines 11-14 of the interval hull example in Sec. 3.7.1.

3.8 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. One can also use the standard MATLAB `LineSpec` settings for the border of reachable sets.



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

(d) `plot(Z,[1,2],'lightgray')` (e) `plot(Z,[1,2],'darkgray')` (f) `plot(Z,[1,2],'blackEdgeThick')`

Figure 15: Selection of different plot styles.

4 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. 5.2. CORA supports the following matrix set representations:

- Matrix polytope (Sec. 4.1)
- Matrix zonotope (Sec. 4.2); specialization of a matrix polytope.
- Interval matrix (Sec. 4.3); specialization of a matrix zonotope.

For each matrix set representation, the conversion to all other matrix set computations is implemented. Of course, conversions to specializations are realized in an over-approximative way, while the other direction is computed exactly. Note that we use the term *matrix polytope* instead of *polytope matrix*. The reason is that the analogous term *vector polytope* makes sense, while *polytope vector* can be misinterpreted as a vertex of a polytope. We do not use the term *matrix*

interval since the term *interval matrix* is already established. Important operations for matrix sets are

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

4.1 Matrix Polytopes

A matrix polytope is analogously defined as a V-polytope (see Sec. 3.5):

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

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. 3.5).

We support the following methods for polytope matrices:

- `display` – standard method, see Sec. 4.
- `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. 4.
- `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. 4 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. 4. 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. 4.

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).

4.1.1 Matrix Polytope Example

The following MATLAB code demonstrates some of the introduced methods:

```
P1{1} = [1 2; 3 4]; % 1st vertex of matrix polytope P1
P1{2} = [2 2; 3 3]; % 2nd vertex of matrix polytope P1
matP1 = matPolytope(P1); % instantiate matrix polytope P1

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

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

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

This produces the workspace output

```
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
```

4.2 Matrix Zonotopes

A matrix zonotope is defined analogously to zonotopes (see Sec. 3.1):

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

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. [3]).

We support the following methods for zonotope matrices:

- `concatenate` – concatenates the center and all generators of two matrix zonotopes.
- `display` – standard method, see Sec. 4.

- `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. 4.
- `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. 4 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. 4) 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. 4.
- `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).

4.2.1 Matrix Zonotope Example

The following MATLAB code demonstrates some of the introduced methods:

```
Zcenter = [1 2; 3 4]; % center of matrix zonotope Z1
Zdelta{1} = [1 0; 1 1]; % generators of matrix zonotope Z1
matZ1 = matZonotope(Zcenter, Zdelta); % instantiate matrix zonotope Z1

Zcenter = [-1 2; 2 -1]; % center of matrix zonotope Z2
Zdelta{1} = [0 0.5; 0.5 0]; % generators of matrix zonotope Z2
matZ2 = matZonotope(Zcenter, Zdelta); % instantiate matrix zonotope Z2

matZ3 = matZ1 + matZ2 % perform Minkowski addition and display result
matZ4 = matZ1 * matZ2 % compute multiplication of and display result

intZ = intervalMatrix(matZ1) % compute interval matrix and display result
```

This produces the workspace output

```
dimension:
     2

nr of generators:
     2

center:
     0     4
     5     3

generators:
     1     0
     1     1

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

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

nr of generators:
     3

center:
     3     0
     5     2

generators:
```

```
    1.0000    0.5000
    2.0000    1.5000

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

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

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

left limit:
     0     2
     2     3

right limit:
     2     2
     4     5
```

4.3 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. 4.
- `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.
- `intervalhull` – converts an interval matrix to an interval hull.
- `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. 4 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. 4. 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. 4.
- `volume` – computes the volume of an interval matrix by computing the volume of the corresponding interval hull.

4.3.1 Interval Matrix Example

The following MATLAB code demonstrates some of the introduced methods:

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

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

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

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

This produces the workspace output

```
dimension:
     2

left limit:
   -1.0000    3.5000
    3.5000    2.0000

right limit:
    1.0000    4.5000
    6.5000    4.0000

dimension:
     2

left limit:
    1.0000   -3.0000
   -0.5000   -3.0000

right limit:
    5.0000    3.0000
   10.5000    7.0000

dimension:
     2

nr of generators:
     3

center:
     1     2
     3     4

generators:
     1     0
     0     0

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

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

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

5 Continuous Dynamics

This section introduces various classes to compute reachable sets of continuous 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. 5.1)
- Linear systems with uncertain fixed parameters (Sec. 5.2)
- Linear systems with uncertain varying parameters (Sec. 5.3)
- Linear probabilistic systems (Sec. 5.4)
- Nonlinear systems (Sec. 5.5)
- Nonlinear systems with uncertain fixed parameters (Sec. 5.6)
- Nonlinear differential-algebraic systems (Sec. 5.7)

For each class the same methods are implemented:

- `display`: Displays the parameters of the continuous dynamics in the MATLAB workspace.
- `initReach`: Initializes the reachable set computation.
- `reach`: 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.

5.1 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).

5.1.1 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 [4, 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 $\tilde{\mathcal{U}}_\Delta$.

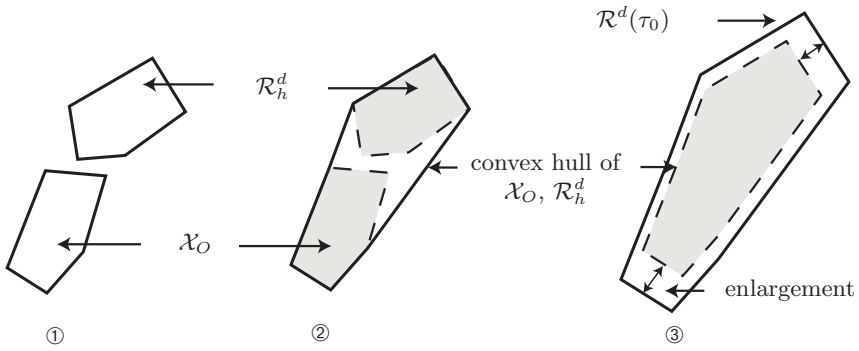


Figure 16: Steps for the computation of an over-approximation of the reachable set for a linear system.

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 [4, 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 [4, Proposition 3.1], which can only be used in combination with [4, 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 [4, 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 [4, Theorem 3.1] is used.

5.2 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. 5.5. 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. 4.
- `B` – set of input matrices $\mathcal{B}$, see (8). The set of matrices can be represented by any matrix set introduced in Sec. 4.
- `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 [4, Theorem 3.2].
- `mappingMatrixSet` – set of exponential matrices, see Sec. 4.
- `E` – remainder of matrix exponential computation.
- `F` – uncertain matrix to bound the error for time interval solutions, see e.g. [4, Proposition 3.1].
- `inputF` – uncertain matrix to bound the error for time interval solutions of inputs, see e.g. [4, Proposition 3.4].
- `inputCorr` – additional uncertainty of the input solution if origin is not contained in input set, see [4, Equation 3.9].
- `Rinput` – reachable set of the input solution, see Sec. 5.1.
- `Rtrans` – reachable set of the input u_c, see Sec. 5.1.
- `RV` – reachable set of the input $\tilde{\mathcal{U}}_\Delta$, see Sec. 5.1.
- `sampleMatrix` – possible matrix $\hat{A}$ such that $\hat{A} \in \mathcal{A}$.

5.2.1 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 [4, 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].

5.3 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. 4. The provided methods of the class are identical to the ones in Sec. 5.2, 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. 4.
- `B` – set of input matrices $\mathcal{B}$, see (8). The set of matrices can be represented by any matrix set introduced in Sec. 4.
- `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 [4, Theorem 3.2].
- `mappingMatrixSet` – set of exponential matrices, see Sec. 4.
- `power` – tba
- `E` – remainder of matrix exponential computation.
- `F` – uncertain matrix to bound the error for time interval solutions, see e.g. [4, Proposition 3.1].
- `inputF` – uncertain matrix to bound the error for time interval solutions of inputs, see e.g. [4, Proposition 3.4].
- `inputCorr` – additional uncertainty of the input solution if origin is not contained in input set, see [4, Equation 3.9].
- `Rinput` – reachable set of the input solution, see Sec. 5.1.
- `Rtrans` – reachable set of the input u_c, see Sec. 5.1.
- `sampleMatrix` – possible matrix $\hat{A}$ such that $\hat{A} \in \mathcal{A}$.

5.4 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 $\bar{f}_{\mathbf{X}}(x, [0, r]) = \sup\{\bar{f}_{\mathbf{X}}(x, t) | t \in [0, r]\}$.

5.4.1 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. 5.1.1. 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. 5.1.
- `tie` (**t**ime **i**nterval **e**rror – computes the `tie` similarly as for linear systems in Sec. 5.1.
- `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].

5.5 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 [4, 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.

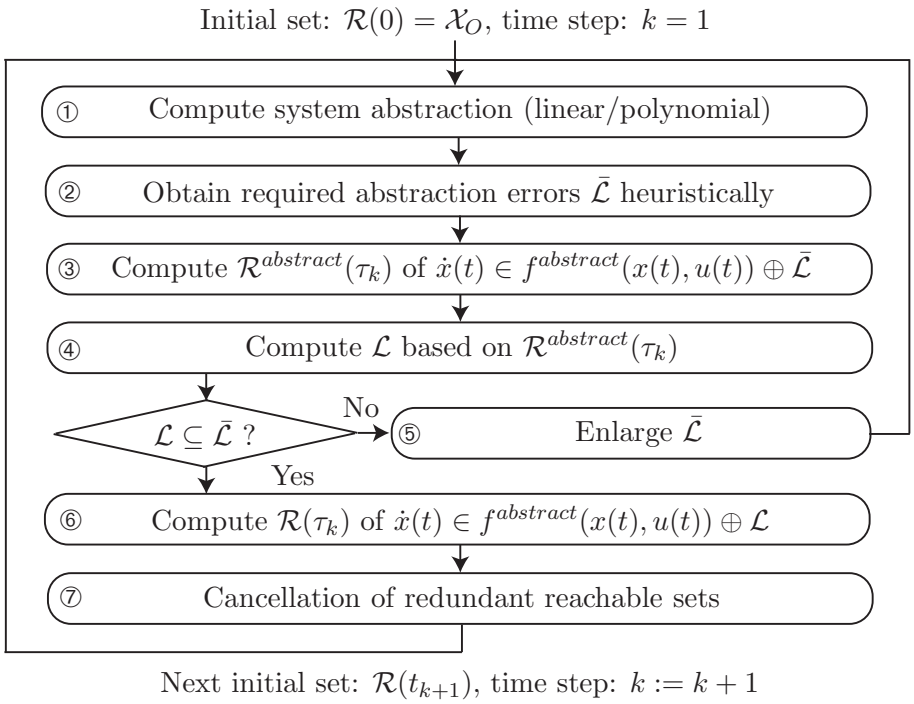


Figure 17: Computation of reachable sets – overview.

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

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

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.

5.5.1 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 arithmatic according to [5, 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].

5.6 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.

5.7 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. 5.6. 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 [5, 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 [5, 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.

5.8 Continuous Dynamics by Example

show some examples here.

6 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[8]. 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. 9), 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 $HA = (\mathcal{V}, v^0, \mathcal{X}, \mathcal{X}^0, \mathcal{U}, \mathcal{P}, \mathtt{inv}, \mathtt{T}, \mathtt{g}, \mathtt{h}, \mathtt{f})$, 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 $\mathcal{X}^0 \subseteq \mathtt{inv}(v^0)$.
- the continuous input space $\mathcal{U} \subseteq \mathbb{R}^m$.
- the parameter space $\mathcal{P} \subseteq \mathbb{R}^p$.
- the mapping[9] $\mathtt{inv}$: $\mathcal{V} \to 2^{\mathcal{X}}$, which assigns an invariant $\mathtt{inv}(v) \subseteq \mathcal{X}$ to each location v.
- the set of discrete transitions $\mathtt{T} \subseteq \mathcal{V} \times \mathcal{V}$. A transition from $v_i \in \mathcal{V}$ to $v_j \in \mathcal{V}$ is denoted by (v_i, v_j).
- the guard function $\mathtt{g} : \mathtt{T} \to 2^{\mathcal{X}}$, which associates a guard set $\mathtt{g}((v_i, v_j))$ for each transition from v_i to v_j, where $\mathtt{g}((v_i, v_j)) \cap \mathtt{inv}(v_i) \neq \emptyset$.
- the jump function $\mathtt{h} : \mathtt{T} \times \mathcal{X} \to \mathcal{X}$, which returns the next continuous state when a transition is taken.
- the flow function $\mathtt{f} : \mathcal{V} \times \mathcal{X} \times \mathcal{U} \times \mathcal{P} \to \mathbb{R}^{(n)}$, which defines a continuous vector field for the time derivative of x: $\dot{x} = \mathtt{f}(v, x, u, p)$.

The invariants $\mathtt{inv}(v)$ and the guard sets $\mathtt{g}((v_i, v_j))$ 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 $\mathtt{inv}(v)$. When the transition from the previous location v_i to the next location v_j is taken, the system state is updated according to the jump function and the continuous evolution within the next invariant.

Because the reachability of discrete states is simply a question of determining if the continuous reachable set hits certain guard sets, the focus of CORA is on the continuous reachable sets. Clearly, as for the continuous systems, the reachable set of the hybrid system has to be over-approximated in order to verify the safety of the system. An illustration of a reachable set of a hybrid automaton is given in Fig. 18.

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

[9] $2^{\mathcal{X}}$ is the power set of $\mathcal{X}$.

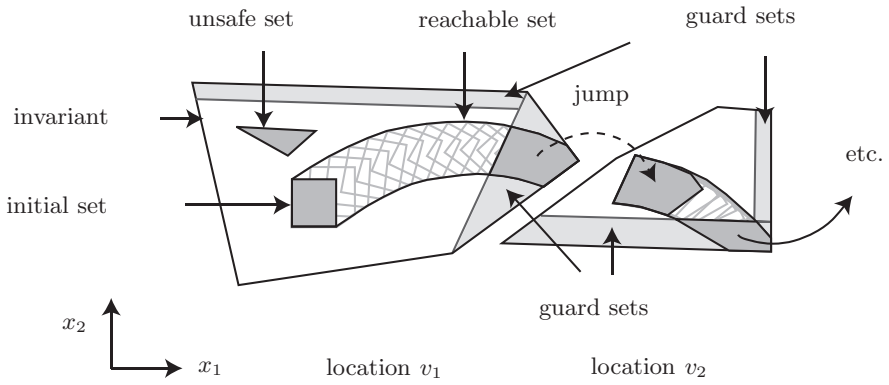


Figure 18: Illustration of the reachable set of a hybrid automaton.

6.1 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.

6.2 Location

Each location consists of:

- `invariant` – specified by a set representation of Sec. 3.
- `transitions` – cell array of objects of the class `transition`.
- `contDynamics` – specified by a continuous dynamics of Sec. 5.
- `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.

6.3 Transition

Each transition consists of

- `guard` – specified by a set representation of Sec. 3.
- `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').

7 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.

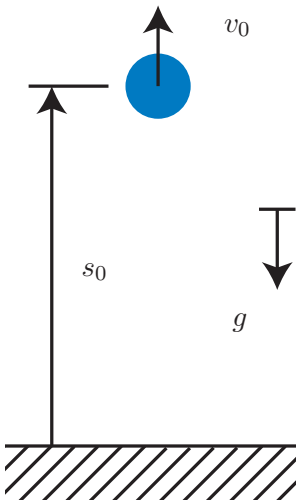
8 Options for Reachability Analysis

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).
- `isHybrid` – flag whether the considered system is hybrid (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.

9 Bouncing Ball Example

We demonstrate the syntax of CORA for the well-known bouncing ball example, see e.g. [23, Section 2.2.3]. Given is a ball in Fig. 19 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. 6 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 19: Bouncing ball.

The MATLAB code that implements the simulation and reachability analysis of the bouncing ball example is:

```
function bouncingBall()

%set options---------------------------------------------------------------
options.x0 = [1; 0]; %initial state for simulation
options.R0 = zonotope([options.x0, diag([0.05, 0.05])]); %initial set
options.startLoc = 1; %initial location
options.finalLoc = 0; %0: no final location
options.tStart = 0; %start time
options.tFinal = 5; %final time
options.timeStepLoc{1} = 0.05; %time step size in location 1
options.taylorTerms = 10;
options.zonotopeOrder = 20;
options.polytopeOrder = 10;
options.errorOrder=2;
options.reductionTechnique = 'girard';
options.isHybrid = 1;
options.isHyperplaneMap = 0;
options.enclosureEnables = [5]; %choose enclosure method(s)
options.originContained = 0;
options.polytopeType = 'mpt';
%--------------------------------------------------------------------------

%specify hybrid automaton--------------------------------------------------
%define large and small distance
dist = 1e3;
eps = 1e-6;

A = [0 1; 0 0]; % system matrix
B = eye(2); % input matrix
linSys = linearSys('linearSys',A,B); %linear continuous system

inv = intervalhull([-2*eps, dist; -dist, dist]); %invariant
guard = intervalhull([- eps, 0; -dist, 0]); %guard set
reset.A = [0, 0; 0, -0.75]; reset.b = zeros(2,1); %reset
trans{1} = transition(guard,reset,1,'a','b'); %transition
loc{1} = location('loc1',1,inv,trans,linSys); %location
HA = hybridAutomaton(loc); % select location for hybrid automaton
```

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

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

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

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

%choose projection and plot--------------------------------------------------
options.projectedDimensions = [1 2];
options.plotType = 'b';
plot(HA,'reachableSet',options); %plot reachable set
plot(options.R0,options.projectedDimensions,'blackFrame'); %plot initial set
plot(HA,'simulation',options); %plot simulation
%------------------------------------------------------------------------
```

The reachable set and the simulation are plotted in Fig. 20 for a time horizon of $t_f = 5$ seconds.

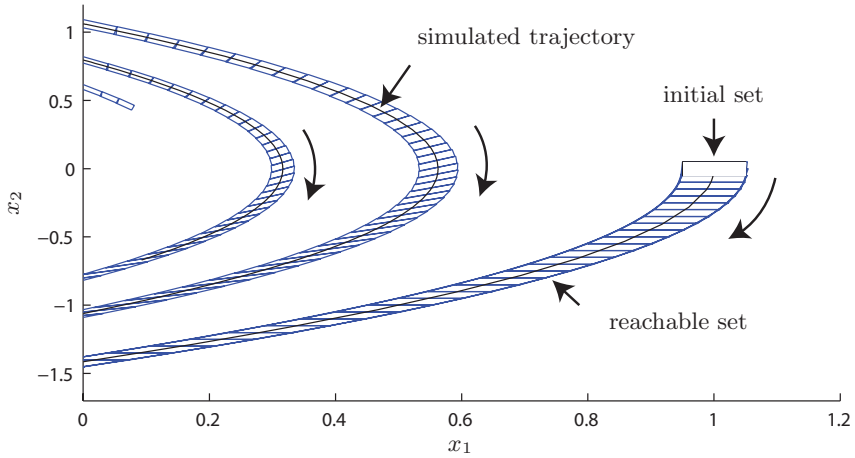


Figure 20: Illustration of the reachable set of the bouncing ball. The black box shows the initial set and the black line shows the simulated trajectory.

10 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/`.

11 Conclusions

CORA is a toolbox for the implementation of prototype reachability analysis algorithms in MATLAB. The software is modular and is organized into four main categories: vector set representations, matrix set representations, continuous dynamics, and hybrid dynamics. CORA includes novel algorithms for reachability analysis of nonlinear systems and hybrid systems with a special focus on scalability; for instance, a power network with more than 50 continuous state variables has been verified in [24]. The efficiency of the algorithms used means it is even possible to verify problems online, i.e. while they are in operation [25].

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.

References

[1] 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.

[2] M. Althoff, "An introduction to cora 2015," in *Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems*, 2015.

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

[4] 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.

[5] 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.

[6] 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.

[7] S. M. Rump, *Developments in Reliable Computing*. Kluwer Academic Publishers, 1999, ch. INTLAB - INTerval LABoratory, pp. 77–104.

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

[24] 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.

[25] M. Althoff and J. M. Dolan, "Online verification of automated road vehicles using reachability analysis," *IEEE Transactions on Robotics*, vol. 30, no. 4, pp. 903–918, 2014.