juliacon

LazySets.jl: Scalable Symbolic-Numeric Set Computations

Marcelo Forets[1,*] and Christian Schilling[2,*]

[1]Universidad de la República, Uruguay
[2]University of Konstanz, Germany

ABSTRACT

LazySets.jl is a Julia library that provides ways to symbolically represent sets of points as geometric shapes, with a special focus on convex sets and polyhedral approximations. LazySets provides methods to apply common set operations, convert between different set representations, and efficiently compute with sets in high dimensions using specialized algorithms based on the set types. LazySets is the core library of JuliaReach, a cutting-edge software addressing the fundamental problem of reachability analysis: computing the set of states that are reachable by a dynamical system from all initial states and for all admissible inputs and parameters. While the library was originally designed for reachability and formal verification, its scope goes beyond such topics. LazySets is an easy-to-use, general-purpose and scalable library for computations that mix symbolics and numerics. In this article we showcase the basic functionality, highlighting some of the key design choices.

Keywords

Set propagation, Geometry, Reachability analysis, Hybrid system, Support function, Formal verification

1. Introduction

LazySets.jl is an open-source Julia package for calculus with geometric sets of points in Euclidean space. For a visual example showing the sets of states reachable by the Lotka-Volterra equations, see Fig. 1. The package provides solutions to represent sets, perform calculations on them, and combine them via set operations to form new sets. A key aspect of LazySets is that set operations can be applied concretely, meaning that a computation is invoked, or lazily, meaning that the computation is delayed. Based on geometric concepts, LazySets can evaluate queries on the lazy set representation, which enables efficient operation in very high dimensions that is not possible when applying the operations concretely.

LazySets aims to be a flexible and scalable library. It provides specialized representations for various common classes of sets and ways for interacting with these sets, and is able to work with complex set constructs by use of the support-function calculus.[1] Flexibility is achieved by implementing generic algorithms that apply to multiple types of sets, and interoperability is achieved by connecting all set types through common interfaces. Efficiency is achieved by adding special-case implementations where applicable; set operations are often binary functions and Julia's multiple dispatch

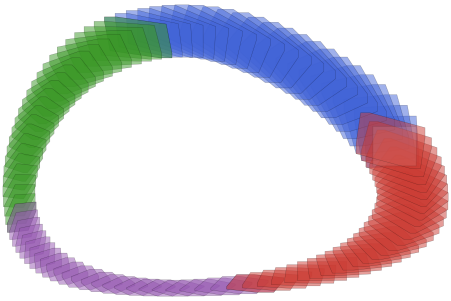


Fig. 1: Example of set propagation using LazySets.

greatly simplifies the choice of the most efficient implementation for a given combination of sets. LazySets is designed to work with very high-dimensional sets but also provides specialized methods for one- and two-dimensional sets. Finally, LazySets is well integrated with the Julia ecosystem for scientific computing [6], and additional functionality is available upon loading optional packages.

In this article we present the basic functionality of LazySets, starting with common sets and operations in Section 2. More advanced topics on type composition and conversion are introduced in sections 3 and 4. Several applications are included in Section 5. We provide references and comment on related libraries in Section 6. Installation instructions can be found in Appendix A. Background mathematical definitions are included in Appendix B. The code to reproduce the figures can be found in Appendix C.

2. Basic sets and operations

LazySets offers support for *convex* and *non-convex* sets. Intuitively, a set X is convex if one can draw a straight line segment between any two points in X without leaving X (see Appendix B.1 for a formal definition). This explains why optimization over a convex set is efficient. Convex sets enjoy several other attractive properties, and many important geometric shapes are convex.

2.1 Constructing sets

Two basic sets are the *hyperplane*

$$\{x \in \mathbb{R}^n \mid a^T x = b\},$$

which is parametric in a vector $a \in \mathbb{R}^n$ and a scalar $b \in \mathbb{R}$, and the *half-space* (or *linear constraint*)

$$\{x \in \mathbb{R}^n \mid a^T x \leq b\},$$

which consists of all points on one side of the corresponding hyperplane. In LazySets these sets are constructed from a and b. For

*Both authors contributed equally.
[1]Complementary background is included in the Appendix.

Table 1. : Set operations available in LazySets. The result of the last three operations is generally not convex even if used with convex operands. For binary operations (marked with $\cdot^b$) there is also an n-ary lazy version with the suffix `Array`, e.g., `MinkowskiSumArray`. Unicode symbols (as mentioned in the column "Short form") are entered in the Julia REPL by typing the LaTeX command (e.g.: `\oplus` for $\oplus$) followed by pressing the "Tab" key. See Appendix B.2 for some central definitions.

Operation name	Math form	Lazy function (constructor)	Short form	Concrete function
Minkowski sum^b	$X \oplus Y$	`MinkowskiSum`	`+`, `\oplus`	`minkowski_sum`
Intersection^b	$X \cap Y$	`Intersection`	`\cap`	`intersection`
Cartesian product^b	$X \times Y$	`CartesianProduct`	`*`, `\times`	`cartesian_product`
Convex hull^b	$CH(X \cup Y)$	`ConvexHull`	`CH`	`convex_hull`
Symmetric interval hull	$\boxdot(X)$	`SymmetricIntervalHull`	`\boxdot`	`symmetric_interval_hull`
Linear map	AX	`LinearMap`	`*`	`linear_map`
Exponential map	$e^A X$	`ExponentialMap`		`exponential_map`
Translation	$X + b$	`Translation`	`+`	`translate`
Affine map	$AX + b$	`AffineMap`	`*` and `+`	`affine_map`
Reset map	$x_i \mapsto c$	`ResetMap`		-
Inverse linear map	$A^{-1}X$	`InverseLinearMap`		-
Bloating	$X \oplus \{x : \|x\| \leq \varepsilon\}$	`Bloating`		-
Union^b	$X \cup Y$	`UnionSet`	`\cup`	-
Complement	X^C	`Complement`		`complement`
Rectified linear unit	$x_i \mapsto \max(x_i, 0)$	`Rectification`		`rectify`

example, the two-dimensional hyperplane $x = 1$ (resp. the half-space $x \leq 1$) are:

```
julia> a = [1.0, 0.0]; b = 1.0

julia> Hyperplane(a, b)
Hyperplane{Float64,Vector{Float64}}([1.0, 0.0], 1.0)

julia> HalfSpace(a, b)
HalfSpace{Float64,Vector{Float64}}([1.0, 0.0], 1.0)
```

Higher-dimensional sets are defined in a similar fashion; for instance, the 100-dimensional half-space $x_1 + \ldots + x_{100} \leq 10$ is:

```
julia> a = fill(1.0, 100); b = 10.0

julia> HalfSpace(a, b)
HalfSpace([1.0, ..., 1.0], 10.0)
```

The most widely used convex sets in various disciplines are (convex) *polyhedra*, which are characterized as the finite intersection of half-spaces. Optimization over a polyhedron corresponds to solving a linear program. In Fig. 2 we show an example of a *polytope* (a bounded polyhedron) with seven (linear) constraints in orange and a half-space in blue.

LazySets contains many (currently: 26) different structs to represent common classes of sets (such as half-spaces). These set types simply expect and store the corresponding parameters to represent the set. For example, the `HalfSpace` stores the vector `a` and the scalar `b`. (There are a few exceptions where the constructor performs normalization by default, e.g., `HPolygon`, representing a two-dimensional polytope, sorts the constraints by the vectors `a` in counter-clockwise order.) Hence construction is fast and the internal representation is space efficient. For instance, the `BallInf` represents a hypercube specified by the center vector $c \in \mathbb{R}^n$ and the radius $r \in \mathbb{R}$. In n dimensions, a hypercube has 2^n vertices, but creating an 1,000-dimensional `BallInf` is instantaneous.

```
julia> @time BallInf(zeros(1000), 1.0)
0.000005 seconds (2 allocations: 7.969 KiB)
```

2.2 Extracting information from sets

Being able to represent sets is not useful by itself because we also want to interact with them. For example, we may want to draw samples from a set. A general approach to do that is rejection sampling, which picks a random point $x \in \mathbb{R}^n$ and checks whether $x \in X$ holds. We can thus use rejection sampling with any set type that implements a membership test.

```
julia> ones(1000) ∈ BallInf(zeros(1000), 1.0)
true
```

Other typical properties that can be checked for sets X and Y are emptiness ($X = \emptyset$; `isempty`), inclusion ($X \subseteq Y$; `issubset`), and having no point in common ($X \cap Y = \emptyset$; `isdisjoint`).

We may also want to obtain information that is encoded in the set representation. For example, we can ask for the list of vertices of a polytope. We have seen that a hypercube is represented by the center and the radius, so the vertices need to be computed on demand.

```
julia> vertices_list(BallInf([1.0, 4.0], 1.0))
4-element Vector{Vector{Float64}}:
 [2.0, 5.0]
 [0.0, 5.0]
 [2.0, 3.0]
 [0.0, 3.0]
```

Equality of sets in the mathematical sense can be checked via `isequivalent` (which by default checks mutual inclusion):

```
julia> X = Interval(-1, 1) × Interval(-1, 1)
CartesianProduct{Float64,
  Interval{...}, Interval{...}}(...)

julia> Y = BallInf(zeros(2), 1.0)

julia> isequivalent(X, Y)
true
```

2.3 Set interfaces

Sometimes the same implementation works for several set types. LazySets uses a hierarchy of abstract types (which we call *interfaces*) to summarize common functionality. For example, `AbstractHyperrectangle` is a supertype of all hyperrectangular set types such as `BallInf` and provides a default implementation to compute the volume. When adding a new set type representing a hyperrectangle, it will automatically use this implementation. The following list is not exhaustive, but should help as a mental model of how the library is organized. Definitions are given from more specific to more general (i.e., less structured).

`AbstractHyperrectangle` : Hyperrectangular sets can be represented by a center vector $c \in \mathbb{R}^n$ and a radius vector $r \in \mathbb{R}^n$. Each $x \in X$ can be written as $x_i = c_i + \xi_i r_i$ for $i = 1, \ldots, n$, for some $\xi_i \in [-1, 1]$. Implementations include intervals (`Interval`), hypercubes (`BallInf`), and the general `Hyperrectangle` .

`AbstractZonotope` : Zonotopic sets are those which admit a representation given by a center $c \in \mathbb{R}^n$ and a finite set of *generators* $g_j \in \mathbb{R}^n$, $j \in 1, \ldots, m$, such that $x \in X$ is can be written as $x = c + \sum_j \xi_j g_j$ for some $\xi_j \in [-1, 1]$. Hyperrectangular sets are also zonotopic, as well as general zonotopes (`Zonotope`).

`AbstractPolyhedron` : A set is called polyhedral if it can be expressed as a finite intersection of half-spaces. Special cases include hyperrectangular and zonotopic sets, as well as more general polytopes (`HPolytope` , `VPolytope`) and also possibly unbounded polyhedra (`HPolyhedron`).

`LazySet` : All convex set types belong to this abstract supertype to prevent type piracy when extending `Base` functions. We are working toward having non-convex sets, such as set unions, in the same type hierarchy as well.

2.4 Set operations

We have seen that we can interact with sets by checking properties. Importantly, we can also apply set operations to sets for constructing new sets. (By default the result is a fresh set and the original set is not manipulated.) For example, one common set operation is to translate (or shift) every element in the set by a constant vector.

```
julia> B1 = BallInf([1.5, 2.0], 1.0)
julia> B2 = translate(B1, [1.5, -1.0])
julia> dump(B2)
BallInf{Float64, Vector{Float64}}
  center: Array{Float64}((2,)) [3.0, 1.0]
  radius: Float64 1.0
```

As seen above, a translation usually preserves the set type. For most operations this is generally not the case. For instance, the intersection of two half-spaces is itself not a half-space but a polyhedron.

```
# intersect {x | x <= 1} and {x | x >= 0}
julia> P = intersection(HalfSpace([1.0], 1.0),
                        HalfSpace([-1.0], 0.0))
julia> typeof(P)
HPolyhedron{Float64, Vector{Float64}}
```

For a complete list of the set operations available in LazySets we refer to Table 1.

3. The LazySets paradigm

Having described how to operate with basic sets, we consider a more fundamental representation problem. We have seen that there exist classes of sets. Some of them, such as polyhedra, are closed under various set operations. That property is convenient because the type of the result is known in advance. The whole class of convex sets is also closed under the operations described in the upper part of Table 1, but we can typically not represent the result with the limited amount of set types in LazySets. This is not a shortcoming of LazySets: you would need infinitely many set representations for all possible combinations! However, we can resort to a simple yet powerful trick to effectively represent the result of the set operations: *lazy representation*. Going a step further, by making both basic set types and operations between sets live in the same abstraction layer (namely always subtyping `LazySet` irrespective whether it is a concrete set or the result of an operation) allows to easily *compose* set computations.

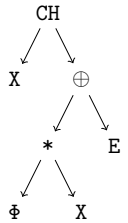
3.1 Type composition

To give an example, the Minkowski sum of a square and a disc (two-dimensional balls in the infinity norm and Euclidean norm) is not representable with a basic type in LazySets. Hence `minkowski_sum` will yield an error. But we can apply the lazy `MinkowskiSum` operation, which itself subtypes `LazySet` . This choice allows for ease of composition.
We wrap the operands in a new object that, by definition, represents the result of the operation, but without actually performing the computation. (This also motivates the name of LazySets).

As an illustrative example, suppose that we are interested in the formula $\Omega_0 = CH(X_0, \Phi X_0 \oplus E_+)$. Such formulas are prevalent in reachability analysis of linear initial-value problems, or nonlinear ones after some form of conservative linearization; see for example [3] and references therein. Given the sets X_0, E_+, and the matrix Φ defined in Appendix C.1, we can write:

```
julia> Ω₀ = CH(X₀, Φ*X₀ ⊕ E₊)
```

Then, Ω_0 is a (nested) *lazy* representation of the operation just as a normal `LazySet` . As such, it can be used for further operations (conversion, approximation, evaluation). The structure of the nested operations is internally represented in the form of a tree, which can be visualized with TreeView.jl as shown in the diagram (right).



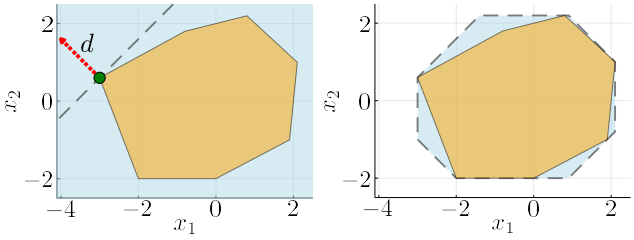


Fig. 2: Supporting hyperplane of the set X along direction d (left), and outer approximation of X using the eight directions of a regular octagon (right).

Lazy operations can be efficiently evaluated, as we describe next.

3.2 Support-function calculus

A standard approach to operate with compact and convex sets $X \in \mathbb{R}^n$ is to use the *support function* [17]. The support function along direction $d \in \mathbb{R}^n$, denoted $\rho(d, X)$, is the maximum of $d^T x$ over all $x \in X$, i.e., the support function describes the (signed) distance of the supporting hyperplane in direction d from the origin. The maximizers are called *support vectors* $\sigma(d, X)$. Intuitively, the support vectors are the extreme points of X in direction d. Basic properties of the support function are given in Appendix B.3.

LazySets offers `ρ(d, X)` (or `support_function(d, X)`) to compute the support function $\rho(d, X)$, and `σ(d, X)` (or `support_vector(d, X)`) to compute (some) support vector $\sigma(d, X)$. Fig. 2 (left) illustrates the evaluation of the support function over polygon X (orange) along direction $(-1.0, 1.0)^T$.

```
julia> d = [-1.0, 1.0]

julia> ρ(d, X) # or support_function(d, X)
3.6

julia> σ(d, X) # or support_vector(d, X)
[-3.0, 0.6]
```

For various set representations, the support function is known analytically and can be efficiently evaluated numerically. Such cases include hyperrectangular sets and zonotopic sets. For sets with less structure, e.g., if X is a polytope in half-space representation, its support function can be computed by solving a linear program (LP), for which fast and robust solvers exist. But the main advantage of using the support function in LazySets lies in the extensive use of *composition rules* as we describe next.

Consider again the set Ω_0 from the previous section. Suppose that we are interested in the support value of Ω_0 along a given direction $d \in \mathbb{R}^2$. Since the support function distributes over the Minkowski sum, $\rho(d, X \oplus Y) = \rho(d, X) + \rho(d, Y)$ for any pair of sets $X, Y \subseteq \mathbb{R}^n$, and since it holds that $\rho(d, MX) = \rho(M^T d, X)$ for any matrix $M \in \mathbb{R}^{n \times n}$, we can propagate the computation through the operation tree until a concrete set is found, and in many cases, an analytic formula is available. That is precisely what LazySets does, automatically, when we make numeric queries to the lazy set such as its support function along $d = (-1, 1)^T$.

```
julia> d = [-1, 1]

julia> @btime ρ($d, $Ω₀)
117.236 ns (2 allocations: 192 bytes)
-0.8
```

We also note that the evaluation of the support function is lazy in the sense that it gives partial information at a reduced cost: In the above computation we have only obtained information for direction d, not for other directions. For many applications it is sufficient to evaluate the support function in only a few directions. To enclose Ω_0 with a bounded set, we can pick a list of *template* directions and use the methods described in the following section.

4. Conversion between set types

LazySets provides ways to convert one set representation to another. If a conversion is not possible due to restrictions in the represented class of sets, LazySets provides ways to obtain approximations. Turning to an approximate but simpler set representation is also interesting for answering questions efficiently that would otherwise be computationally expensive.

4.1 Conversion

LazySets extends Julia's `convert` function for converting between set representations. The first argument is the target type and the second argument is the source set. Below are three mathematically equivalent representations of the interval $X = [0, 1] \subseteq \mathbb{R}$:

```
julia> X = Interval(0, 1)
Interval{Float64,
  IntervalArithmetic.Interval{Float64}}([0, 1])

julia> convert(Hyperrectangle, X)
Hyperrectangle{Float64, Vector{Float64},
  Vector{Float64}}([0.5], [0.5])

julia> convert(Zonotope, X)
Zonotope{Float64, Vector{Float64},
  Matrix{Float64}}([0.5], [0.5])
```

There are even more possibilities, such as representing X as an intersection of half-spaces (try `convert(HPolytope, X)`).

With multiple dispatch it is easy to define less obvious conversions, e.g., to convert the Cartesian product of an interval and a two-dimensional hyperrectangle to a three-dimensional zonotope:

```
julia> X = rand(Interval)

julia> Y = rand(Hyperrectangle, dim=2)

julia> Z = convert(Zonotope, X × Y)
Zonotope{Float64, ...}

julia> dim(Z)
3
```

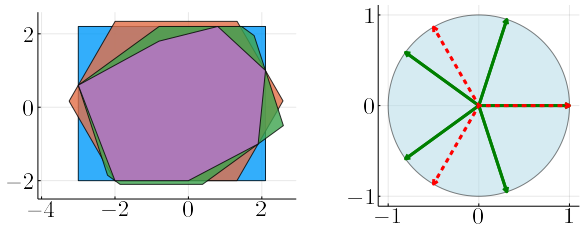


Fig. 3: Left picture: Overapproximation of the polytope from Fig. 2 (purple) with a hyperrectangle (blue) and two zonotopes. The zonotope generators were synthesized from three (red) resp. five (green) polar directions (right). Observe that the approximations are pairwise incomparable.

4.2 Approximation

In many applications we do not require exact results but are content with an approximation. To still give mathematical guarantees, one usually aims for either over- or underapproximations.
We can use the support function to get an overapproximation: For every nonempty compact convex set $X \subseteq \mathbb{R}^n$ and $D \subseteq \mathbb{R}^n$ we have

$$X \subseteq \bigcap_{d \in D} \{d^T x \leq \rho(d, X)\}$$

and equality holds for $D = \mathbb{R}^n$.

LazySets has predefined common template directions such as `OctDirections(2)` for directions normal to a regular octagon in two dimensions. Fig. 2 (left) illustrates the evaluation of overapproximating the set X with octagonal directions, resulting in a polygon with eight constraints. Apart from common fixed template directions there are also options for parametric uniform directions in two (`PolarDirections`) or three (`SphericalDirections`) dimensions or for a custom set of directions (`CustomDirections`).

```
julia> Xoct = overapproximate(X, OctDirections(2))

julia> length(constraints_list(Xoct))
8
```

In two dimensions (or the projection of a higher-dimensional set into two dimensions), LazySets can compute ε-close overapproximations via `overapproximate(X, ε)` , where ε is the specified tolerance. Such overapproximations generally yield an `HPolytope` if X is bounded.

In some applications, we may want to ensure that the result has a specific set type. The smallest bounding box is available by specifying the second argument type. It yields a `Hyperrectangle` , which is more efficient to work with.

```
julia> overapproximate(X, Hyperrectangle)

julia> box_approximation(X) # alias
```

We can use `overapproximate(P, Zonotope, D)` , where `P` is a polytope and `D` is a vector of directions used as candidates for the generators, to obtain a zonotope (in any dimension). We show some example overapproximations in Fig. 3.

5. Ecosystem

The Julia language features a rich ecosystem for scientific computing. LazySets uses Requires.jl to add further functionality depending on whether some external packages are loaded. Here we quickly list various examples, but the list is not exhaustive. From three-dimensional visualizations to reachability analysis using non-convex set representations, there is a broad spectrum of features that are available loading optional dependencies.

Some LazySets functionality prints instructive error messages when the corresponding packages are not loaded but are required by the code. For example, the conversion between constraint and vertex representation of general polytopes in dimension higher than two requires the optional package Polyhedra.jl:

```
julia> X = rand(VPolytope, dim=4)

julia> constraints_list(X)
ERROR: AssertionError: package 'Polyhedra' not
  loaded (it is required for executing
  `default_polyhedra_backend`)

# fix the error by loading the optional package
julia> import Polyhedra

julia> constraints_list(X)
12-element Vector{HalfSpace{Float64, ...}}:
...
```

5.1 Plotting three-dimensional sets and projections

While writing this article, we received the following question:

> We need to plot polyhedra given in a form like this:
>
> 2*x1 >= 0 & 3*x2+1.7*x3 >=0
>
> Is there a way to plot a 2D projection with LazySets?

Yes, there is! We replied with the script below. The script nicely illustrates the interaction of LazySets with the Julia ecosystem. We use Symbolics.jl for reading the polyhedron in symbolic form, Polyhedra.jl and CDDLib.jl for projecting the polyhedron, and Plots.jl to plot the two-dimensional projection.

```
using Symbolics
import Polyhedra, CDDLib, Plots

# projected polyhedron from symbolic constraints
vars = @variables x1, x2, x3  # create symbols
P = HPolyhedron([2*x1>=0, 3*x2+1.7*x3>=0], vars)
Q = project(P, [2, 3])  # 2D projection (x2 and x3)

Plots.plot(Q)  # plot the 2D projection
```

We can also plot the three-dimensional set with Makie.jl. The Makie plot and the two-dimensional projection are shown in Fig. 4.

```
import GLMakie

# intersection with a bounding box
B = BallInf(zeros(3), 5.0)
R = intersection(P, B)

plot3d(R)  # plot in three dimensions
```

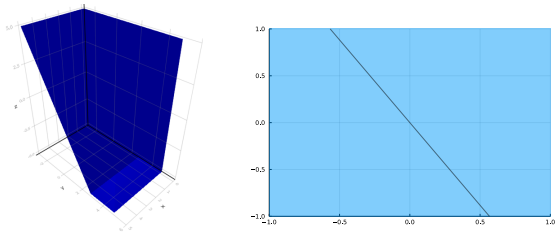


Fig. 4: Three-dimensional polyhedron plotted using Makie (left) and a two-dimensional projection obtained by Polyhedra with CDDLib (right).

5.2 Generic numbers and automatic differentiation

LazySets types are parametric in the number type. Hence it is simple to use custom number types. Besides rationals and arbitrary-precision floating-point numbers, it is possible to make rigorous floating-point calculations using IntervalArithmetic.jl. Moreover, LazySets features a mechanism to globally tune the numeric tolerances used in floating-point operations. To do so, use `set_atol`, `set_rtol`, and `set_ztol` (for absolute, relative, and comparison-with-zero tolerance) respectively. All set functions have been carefully designed to consistently use the specified tolerance and preserve it during operations.

Questions from users, bug reports, and feature requests are available in the issue tracker. One question we received was:

> Using ForwardDiff.jl to get the gradient of [...] throws the following error:
> `ERROR: default tolerance for numeric type ForwardDiff.Dual{...} is not defined`

The solution was surprisingly simple: extending the LazySets tolerance mechanism to work with dual numbers fixed the error.

```
import ForwardDiff
import LazySets.default_tolerance

default_tolerance(::Type{<:ForwardDiff.Dual}) =
default_tolerance(Float64)
```

With this code, the user could differentiate through the formula `area(intersection(X, Y))`, which computes the area of the intersection between sets X and Y.

5.3 Taylor models as an example of non-convex sets

LazySet is not limited to convex set representations. Apart from operating with set unions (`UnionSet`), which are generally non-convex, the library offers functionality to handle intrinsically non-convex set representations. Examples include star sets (`AbstractStar`), polynomial zonotopes (`PolynomialZonotope`), and Taylor models. The latter representation is available in the package TaylorModels.jl [5, 4], which LazySets interacts with.

Taylor models are, informally, sets defined by the image of a polynomial map with interval remainders. To illustrate, we consider using LazySets to convert, or evaluate the range, of a Taylor model (the conversion is formally explained in [21]). It is easy to operate with Taylor models in LazySets (the definition of `vTM` for the linear and nonlinear case is given in Appendix C.4):

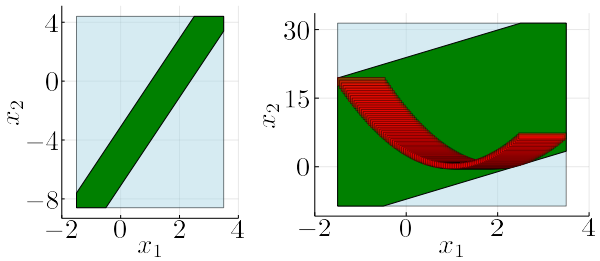


Fig. 5: Exact conversion of a linear Taylor model to a zonotope against an inexact evaluation using interval arithmetic (right). Approximate conversion of a non-linear Taylor model using a single zonotope, and range evaluation with interval arithmetic by splitting its domain into 100 intervals (right).

```
using TaylorModels

# approximate a vector of Taylor models
# with a zonotopes
overapproximate(vTM, Zonotope)

# approximate a vector of Taylor models
# with a hyperrectangle
overapproximate(vTM, Hyperrectangle)
```

In the case when the Taylor model is linear, the conversion is exact, because the zonotope stores linear dependencies between variables. Hence, as is shown in Fig. 5 (left), the zonotope approximation (`Z`) is better than directly using interval arithmetic to evaluate the range (`H`), and queries about the zonotope provide quick information about the exact set, more accurate than the box approximation:

```
julia> d = [-0.35, 0.93]

julia> ρ(d, Z)
3.217

julia> ρ(d, H)
4.617000000000001
```

If the Taylor model contains nonlinear terms, the zonotope provides an enclosure; a comparison against an evaluation using interval arithmetic with many small boxes is shown in Fig. 5 (right).

We also mention that LazySets can be combined with the powerful interval constraint programming approach by simply loading the optional dependency IntervalConstraintProgramming.jl.

5.4 Reachability applications

LazySets is the core library of JuliaReach, a Julia ecosystem to perform reachability analysis of dynamical systems. JuliaReach builds on sound scientific approaches and was, in two occasions (2018 and 2020), the winner of the annual friendly competition on Applied Verification for Continuous and Hybrid Systems (ARCH-COMP).[2]

Reachability techniques are implemented in the JuliaReach package ReachabilityAnalysis.jl, which uses LazySets at its core for dealing with sets, including the computation of flowpipes (union of reachable states). We include two examples. In Fig. 6, a flowpipe for the vertical velocity of an 8-dimensional helicopter model

[2] JuliaReach also participated in the 2021 edition, but the report is not published yet.

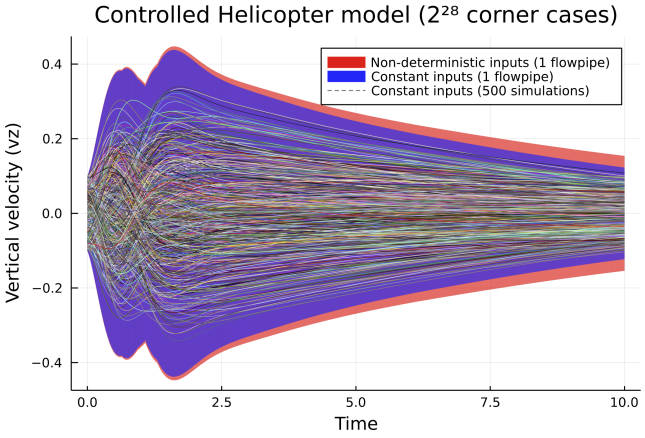


Fig. 6: Reachable states for the vertical velocity of the helicopter model with non-deterministic inputs. Five hundred trajectories drawn randomly from the set of initial states are shown on top.

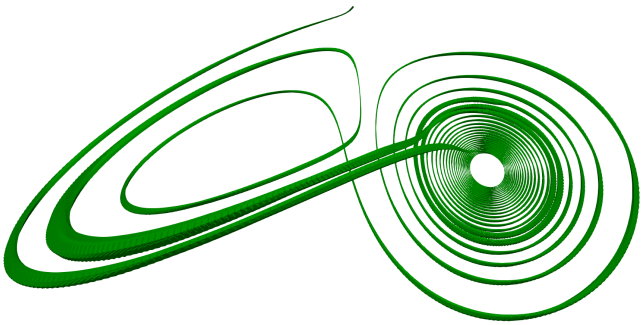


Fig. 7: Zonotope overapproximation of a Taylor-model flowpipe for the Lorenz equations. The plot is rendered with Paraview.

with a 20-dimensional controller from [24] is shown. The model has distributed initial conditions for all variables as well as non-deterministic inputs (input functions can vary arbitrarily within specified bounds). The computation terminates in nearly 50 ms, illustrating the precision and speed of LazySets. It should be noted that the set of initial states has 2^{28} corner cases, thus even in the simpler setting where the inputs are held constant, exhaustive evaluation using simulations is computationally intractable, since it would require 268 million runs.

As a second example, a three-dimensional flowpipe for the well-known Lorenz system is shown in Fig. 7. In this simulation, the initial conditions are defined as a flat hyperrectangle of radius 0.1 along the x coordinate:

```
X0 = Hyperrectangle(low=[0.9, 0.0, 0.0],
                    high=[1.1, 0.0, 0.0])
```

Reachable states are represented using Taylor models, which are then approximated with zonotopes for further computations. In this case we have exported the LazySets objects to a VTK file using the WriteVTK.jl optional dependency, and the rendering was performed by the open-source visualization tool Paraview.

5.5 Parametrization and custom array types

For set types that contain array fields, we use type parameters. Hence it is possible to instantiate LazySets types with any custom array. Typically, such special arrays (e.g. dense, sparse, static) are used in applications that require high performance.

For example, static arrays (where the size of the array can be determined from the type) are preferable for efficient set computations with "small" arrays and are available upon loading StaticArrays.jl:

```
julia> using StaticArrays

# random ten-dimensional zonotope
julia> Z = rand(Zonotope, dim=10, num_generators=30)

# convert to static arrays
julia> Zs = Zonotope(SVector{10, Float64}(Z.center),
SMatrix{10, 30, Float64}(Z.generators))
julia> d = ones(10)
julia> ds = SVector{10, Float64}(d)

# using normal arrays
julia> @btime ρ($d, $Z)
145.290 ns (0 allocations: 0 bytes)
72.68047192978314

# using static arrays
julia> @btime ρ($ds, $Zs)
44.899 ns (0 allocations: 0 bytes)
72.68047192978314
```

Custom arrays can also be used to express knowledge about the *structure* of the set, using only type information. For instance, an axis-aligned half-space can be defined such that its normal vector is a `LazySets.SingleEntryVector` , i.e., a vector with a single non-zero element. Then, the concrete intersection with a hyperrectangular set can be computed very efficiently:

```
# half-space with a normal array
julia> @btime intersection($X, $Hvec)
419.424 μs (1799 allocations: 177.83 KiB)

# half-space with a specialized array
julia> @btime intersection($X, $Hsev)
376.059 ns (13 allocations: 1.08 KiB)
```

5.6 Interoperability with other languages

It is not uncommon that scientists beginning to use Julia are familiar with other languages, specially with the Python programming language. Below we show how to use pyjulia for working with LazySets types *from Python*. We can see that it is not necessary to use Julia objects everywhere; NumPy arrays can also be used, making the interoperability between Julia and Python effortless.

```
$ python3 -m pip install --user julia

$ python3

>>> import julia
>>> julia.install()  # only once

>>> from julia import Base, LazySets
>>> from julia.LazySets import BallInf, volume

>>> B = BallInf(Base.zeros(3), 1.0)
>>> volume(B)
8.0

>>> import numpy as np
>>> c = np.array([0.0, 0.0, 0.0])
>>> B = BallInf(c, 1.0)
>>> volume(B)
8.0
```

6. Conclusion and Perspectives

We conclude with a brief discussion of the past, present, and future development perspectives of LazySets.

6.1 Origin of LazySets and current applications

LazySets has its origins in a tool for reachability analysis of linear dynamical systems, using a compositional approach based on reducing high-dimensional lazy set representations into a sequence of low-dimensional projections that can be computed efficiently [9]. This method presented the first approach to formally verify a 10,000-dimensional benchmark from control engineering. The reachability tool has since been rewritten in ReachabilityAnalysis.jl. A preliminary exposition of these tools appeared in [7].

We have recently applied LazySets to compute reachable states for linear wave propagation problems and heat transfer problems [11]. The scalability of the approach relies on exploiting the structure of linear systems through the support-function calculus and lazy evaluation. Moreover, linear systems can be embedded in algorithms to analyze nonlinear systems [13]. Further case studies and comparison with other state-of-the-art tools can be found in [2, 15]. LazySets has also been applied to the challenging domain of hybrid systems (systems with mixed discrete-continuous dynamics) for set propagation [8] and synthesis [25]. Such problems require switching between different set representations and handling intersections efficiently and accurately. Timed systems with non-deterministic events have been considered in [12]; the approach is able to handle a large number of sets (100 million sets in zonotope representation), and it was shown to be an order of magnitude faster than competing tools [2].

Besides reachability analysis, LazySets can be used for other purposes. The tool has been adopted in a review article in the context of propagating sets through neural networks [20], and new tools use LazySets for verification, e.g., NeuralNetworkAnalysis.jl [21] and OVERTVerify.jl [23]. Other Julia packages using LazySets functionality include computation of invariant sets in InvariantSets.jl, ray tracing for geometric optics in OpticSim.jl, astronomical photometry in Photometry.jl, thin film simulations in Swalbe.jl, and linear algebra with interval matrices in IntervalLinearAlgebra.jl.

6.2 Related libraries

There are few publicly available libraries with a similar aim as LazySets. All these libraries are used in the context of reachability analysis. HyPro is a C++ library for concrete representation and manipulation of sets such as convex polytopes and Taylor models and also offers a support-function representation of set operations [22]. CORA is an actively developed Matlab library centered around zonotopes and contains implementations of zonotope bundles, matrix zonotopes, and polynomial zonotopes [1]. The ellipsoidal toolbox is a Matlab library for ellipsoids [16]. SpaceEx is a C++ library that established the use of the support function in reachability analysis, but it is not open source [14].

Finally, we mention other related Julia packages with a different aim. As mentioned in Section 5, Polyhedra.jl [19] provides an interface for polyhedral computations and the double-description method; hence it complements nicely the lazy features available in our library. GeometryBasics.jl offers standard geometry types, creating a basis for graphics/plotting in Julia. The more recent package Meshes.jl is specialized on efficient and pure-Julia implementations of computational geometry and meshing algorithms. Another Julia package for describing domains in Euclidean space is DomainSets.jl; this package is being adopted for modeling PDE (partial differential equation) domains with ModelingToolkit.jl. Optimization is another scientific field where sets play a major role, with great contributions from Julia developers. MathOptInterface.jl [18] is at the core of JuMP.jl [10], Julia's mainstream modeling language for mathematical optimization. Applications include set programming in SetProg.jl and set distances in MathOptSetDistances.jl.

6.3 LazySets in numbers

At the time of writing (LazySets-v1.52.1), the package consists of 125 source files with almost 26k lines of code (LOC), 66 test files with over 5k LOC, and 66 documentation files (markdown) with over 4k LOC. It is maintained by the authors of this article[3]. The project has received contributions from 13 other people. LazySets was used in 19 research articles.

6.4 Future work

Set computations often do not allow for typical tricks you would expect to see in a Julia package. For instance, when working with generic polyhedra, there is very little structure, so most information cannot be statically inferred and needs to be computed from the concrete values (such as whether the polyhedron is empty). That is why there are so many set types: to bring in more structure for algorithms and dispatch. Another challenge in set computations is to preserve type stability: in some cases, the output set type cannot be predicted in advance.

While many algorithms are already optimized, some functions still use a suboptimal, generic fallback. We are interested to identify and fix such cases. In our experience, one can often obtain speed-ups within several orders of magnitude by adding new methods.

Another direction is the use of trait-based dispatch, which may be useful as a workaround for limitations of the Julia type system, e.g., that it does not allow for multiple inheritance. Expressing properties of sets that fall outside the established LazySets type hierarchy would allow for even further flexibility.

Our next aimed milestone is proper support of non-convex set representations. While functionality to operate with such set represen-

[3] `@mforets` and `@schillic` handles on github.com, respectively.

tations is already available, the interoperability between convex and non-convex sets has room for improvement.

Finally, LazySets has a solid documentation of its API by an extensive use of docstrings and uses Documenter.jl for its online documentation. We plan to add more introductory examples and tutorials for first-time users.

Acknowledgments

First, we thank all those of have contributed to LazySets[4]. We are specially thankful to the following people for discussions and contributions at various stages of this work (sorted alphabetically): Tomer Arnon, Luis Benet, Aadesh Deshmukh, Goran Frehse, Daniel Freire, Bruno Garate, Ander Gray, Sebastian Guadalupe, Nikolaos Kekatos, Benoit Legat, Jorge Pérez Zerpa, Kostiantyn Potomkin, David P. Sanders, Frederic Viry, Ueli Wechsler and Peng Yu. Finally, we thank the people behind Julia Seasons of Contributions and Google Summer of Code programs for their financial support of several students to further develop LazySets. The first author is partly supported by Agencia Nacional de Investigación e Innovación (ANII), Uruguay.

7. References

[1] Matthias Althoff. An introduction to CORA 2015. In *ARCH@CPSWeek*, volume 34 of *EPiC Series in Computing*, pages 120–151. EasyChair, 2015. doi:10.29007/zbkv.

[2] Matthias Althoff, Stanley Bak, Zongnan Bao, Marcelo Forets, Goran Frehse, Daniel Freire, Niklas Kochdumper, Yangge Li, Sayan Mitra, Rajarshi Ray, Christian Schilling, Stefan Schupp, and Mark Wetzlinger. ARCH-COMP20 category report: Continuous and hybrid systems with linear continuous dynamics. In *ARCH*, volume 74 of *EPiC Series in Computing*, pages 16–48. EasyChair, 2020. doi:10.29007/7dt2.

[3] Matthias Althoff, Goran Frehse, and Antoine Girard. Set propagation techniques for reachability analysis. *Annual Review of Control, Robotics, and Autonomous Systems*, 4(1):369–395, 2020. doi:10.1146/annurev-control-071420-081941.

[4] Luis Benet, Marcelo Forets, David P. Sanders, and Christian Schilling. Taylormodels.jl: Taylor models in julia and its application to validated solutions of ODEs. In *SWIM*, 2019.

[5] Luis Benet and David P. Sanders. JuliaIntervals/TaylorModels.jl. `https://github.com/JuliaIntervals/TaylorModels.jl`, 2021. doi:10.5281/zenodo.2613102.

[6] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017. doi:10.1137/141000671.

[7] Sergiy Bogomolov, Marcelo Forets, Goran Frehse, Kostiantyn Potomkin, and Christian Schilling. JuliaReach: a toolbox for set-based reachability. In *HSCC*, pages 39–44. ACM, 2019. doi:10.1145/3302504.3311804.

[8] Sergiy Bogomolov, Marcelo Forets, Goran Frehse, Kostiantyn Potomkin, and Christian Schilling. Reachability analysis of linear hybrid systems via block decomposition. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(11):4018–4029, 2020. doi:10.1109/TCAD.2020.3012859.

[9] Sergiy Bogomolov, Marcelo Forets, Goran Frehse, Frédéric Viry, Andreas Podelski, and Christian Schilling. Reach set approximation through decomposition with low-dimensional sets and high-dimensional matrices. In *HSCC*, pages 41–50. ACM, 2018. doi:10.1145/3178126.3178128.

[10] Iain Dunning, Joey Huchette, and Miles Lubin. JuMP: A Modeling Language for Mathematical Optimization. *SIAM Review*, 59(2):295–320, 2017. doi:10.1137/15M1020575.

[11] Marcelo Forets, Daniel Freire Caporale, and Jorge M Pérez Zerpa. Combining set propagation with finite element methods for time integration in transient solid mechanics problems. *CoRR*, 2021.

[12] Marcelo Forets, Daniel Freire, and Christian Schilling. Efficient reachability analysis of parametric linear hybrid systems with time-triggered transitions. In *MEMOCODE*, pages 1–6. IEEE, 2020. doi:10.1109/MEMOCODE51338.2020.9314994.

[13] Marcelo Forets and Christian Schilling. Reachability of weakly nonlinear systems using Carleman linearization. *CoRR*, 2021.

[14] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable verification of hybrid systems. In *CAV*, volume 6806 of *LNCS*, pages 379–395. Springer, 2011. doi:10.1007/978-3-642-22110-1_30.

[15] Luca Geretti, Julien Alexandre Dit Sandretto, Matthias Althoff, Luis Benet, Alexandre Chapoutot, Xin Chen, Pieter Collins, Marcelo Forets, Daniel Freire, Fabian Immler, Niklas Kochdumper, David P. Sanders, and Christian Schilling. ARCH-COMP20 category report: Continuous and hybrid systems with nonlinear dynamics. In *ARCH*, volume 74 of *EPiC Series in Computing*, pages 49–75. EasyChair, 2020. doi:10.29007/zkf6.

[16] Alexander A. Kurzhanskiy and Pravin Varaiya. Ellipsoidal toolbox (et). In *CDC*, pages 1498–1503, 2006. doi:10.1109/CDC.2006.377036.

[17] Colas Le Guernic. *Reachability analysis of hybrid systems with linear continuous dynamics*. PhD thesis, Université Grenoble 1 - Joseph Fourier, 2009.

[18] Benoît Legat, Oscar Dowson, Joaquim Dias Garcia, and Miles Lubin. Mathoptinterface: a data structure for mathematical optimization problems, 2020.

[19] Benoît Legat, Robin Deits, Gustavo Goretkin, Twan Koolen, Joey Huchette, Daisuke Oyama, and Marcelo Forets. JuliaPolyhedra/Polyhedra.jl: v0.6.16, June 2021. doi:10.5281/zenodo.4993670.

[20] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher A. Strong, Clark W. Barrett, and Mykel J. Kochenderfer. Algorithms for verifying deep neural networks. *Found. Trends Optim.*, 4(3-4):244–404, 2021. doi:10.1561/2400000035.

[21] Christian Schilling, Marcelo Forets, and Sebastian Guadalupe. Verification of neural-network control systems by integrating taylor models and zonotopes. *Submitted.*, 2021.

[22] Stefan Schupp, Erika Ábrahám, Ibtissem Ben Makhlouf, and Stefan Kowalewski. HyPro: A C++ library of state set representations for hybrid systems reachability analysis. In *NFM*, volume 10227 of *LNCS*, pages 288–294, 2017. doi:10.1007/978-3-319-57288-8_20.

[23] Chelsea Sidrane, Amir Maleki, Ahmed Irfan, and Mykel J. Kochenderfer. OVERT: An algorithm for safety verification

[4] The full list of contributors is available on Github.

of neural network control policies for nonlinear systems, 2021.

[24] Sigurd Skogestad and Ian Postlethwaite. *Multivariable feedback control: analysis and design*, volume 2. John Wiley & Sons, 2007.

[25] Miriam García Soto, Thomas A. Henzinger, and Christian Schilling. Synthesis of hybrid automata with affine dynamics from time-series data. In *HSCC*, pages 2:1–2:11. ACM, 2021. doi:10.1145/3447928.3456704.

APPENDIX

A. How to install LazySets.jl

To use LazySets, first install Julia version v1.3 or higher[5]. LazySets is a registered Julia package, and as such, you can install it by activating the `pkg` mode (type `]`, and to leave it, type `<backspace>`), followed by

```
pkg> add LazySets
```

To load the package in a Julia session, do `using`, e.g.

```
julia> using LazySets

julia> HalfSpace([1.0, 0.0], 1.0)
HalfSpace{Float64, Vector{Float64}}([1.0, 0.0], 1.0)
```

The LazySets reference manual is available online at https://juliareach.github.io/LazySets.jl/dev/.

B. Mathematical definitions

B.1 Compact convex sets

Given a set $X \subseteq \mathbb{R}^n$, its *convex hull* is defined as

$$CH(X) = \{\lambda \cdot x + (1-\lambda) \cdot y \mid x, y \in X, \lambda \in [0,1] \subseteq \mathbb{R}\}.$$

A set X is *convex* if it coincides with its convex hull. A set is *closed* if it contains all its boundary points. A set X is *bounded* if there exists a $\delta \in \mathbb{R}$ such that for all $x, y \in X$ it holds that $\|x - y\| \leq \delta$. A set is *compact* if it is closed and bounded.

B.2 Set operations

Given two sets $X, Y \subseteq \mathbb{R}^n$, the *Minkowski sum* is

$$X \oplus Y = \{x + y \mid x \in X, y \in Y\}$$

The *symmetric interval hull* of X is the smallest hyperrectangle that is centrally symmetric in the origin and contains X.
Maps such as the *linear map* AX are applied element-wise:

$$AX = \{Ax \mid x \in X\}$$

B.3 Basic properties of support functions

The support function is a basic notion for approximating convex sets. Let $X \subset \mathbb{R}^n$ be a compact convex set. The support function of X is the function defined as

$$\rho(d, X) := \max_{x \in X} d^{\mathrm{T}} x. \tag{1}$$

We recall the following elementary properties of the support function. For all compact convex sets X and Y in $\mathbb{R}^n$, for all $n \times n$ real matrices M, all scalars λ, and all vectors $d \in \mathbb{R}^n$, we have:

$$\begin{aligned}
&\rho(d, X \oplus Y) = \rho(d, X) + \rho(d, Y) \\
&\rho(d, X \times Y) = \rho(d_1, X) + \rho(d_2, Y) \\
&\rho(d, CH(X \cup Y)) = \max(\rho(d, X), \rho(d, Y)) \\
&\rho(d, MX) = \rho(M^T d, X) \\
&\rho(d, \lambda X) = \rho(\lambda d, X)
\end{aligned}$$

Properties of the support vector (maximizers of (1)) can be found on the LazySets online documentation. Analytic formulas for many important set types are known, allowing for efficient evaluations. The LazySets docstrings contain mathematical explanations and references to the relevant literature (see for example `?Zonotope`).

C. Code used in examples

The complete code for all examples can be found in the repository `http://github.com/JuliaReach/LazySets-JuliaCon21`. In this section we comment on some aspects of the code.

C.1 Code for Fig. 3.1

The set X_0 is a ball in the infinity norm of radius 0.1 centered in $[1, 0]$, the set E_+ is a hyperrectangle centered in the origin, and Φ is a 2×2 matrix defined below. In the context of reachability analysis for linear differential equations [9], the set X_0 corresponds to the initial states, E_+ accouts for bloating terms, and $\Phi = e^{A\delta}$ is the state-transition matrix for some matrix A and time step $\delta > 0$.

```
A = [0 1; -(4π)^2 0]
X₀ = BallInf([1.0, 0.0], 0.1)
δ = 0.025
Φ = exp(A*δ)
  2×2 Matrix{Float64}:
    0.95105652  0.02459079
   -3.88322208  0.95105652

r = [0.05477208, 0.07676220]
E₊ = Hyperrectangle(zeros(2), r)
Ω₀ = CH(X₀, Φ*X₀ ⊕ E₊)
```

C.2 Code for Fig. 2

In Fig. 2, the set X is a polygon in vertex representation. Such a `VPolygon` can be constructed from a vector of points, or simply a matrix where each column corresponds to the coordinates of a point. (For higher-dimensional sets in vertex representation the set type `VPolytope` is used).

```
# two-dimensional polygon in vertex representation
X = VPolygon([-3   -2   0   1   2   0  -0.8;
               0.6  -2  -2  -1   1   2   1.8])
```

The supporting half-space of X is computed by evaluation of the support function along the direction of interest.

[5] Julia binaries can be downloaded from https://julialang.org/downloads/

```
# computing a supporting half-space
d = [-1.0, 1.0]
sf = ρ(d, X)
H = HalfSpace(d, sf)
```

C.3 Code for Fig. 3

The set X is the same as in Fig. 2. We overapproximate it with a box (`Y`) and two zonotopes (`Z` and `W`).

```
Y = box_approximation(X)
Z = overapproximate(X, Zonotope, PolarDirections(3))
W = overapproximate(X, Zonotope, PolarDirections(5))
```

The third argument to `overapproximate` here represents a list of vectors that are used to synthesize the generators of the resulting zonotope. The type `PolarDirections` lazily represents vectors that uniformly cover the unit disc, starting with the vector $(1, 0)^T$.

```
collect(PolarDirections(5))
5-element Vector{Vector{Float64}}:
 [1.0, 0.0]
 [0.30901699437494745, 0.9510565162951535]
 [-0.8090169943749473, 0.5877852522924732]
 [-0.8090169943749475, -0.587785252292473]
 [0.30901699437494723, -0.9510565162951536]
```

C.4 Code for Section 5.3

The linear Taylor models are $p_1(t) = 2 + t$ and $p_2(t) = 0.9 + 3t$ with remainders $I_1 = I_2 = [-0.5, 0.5]$ in the domain $D = [-3, 1]$ and centered at zero. The non-linear case has $p_1(t)$ as in the linear one and $p_2(t) = 0.9 + 3t + 3t^2$ with the same remainders and domain as in the linear case.

```
using TaylorModels
const IA = IntervalArithmetic
const TM1 = TaylorModel1

I = IA.Interval(-0.5, 0.5)
x₀ = IA.Interval(0.0)
D = IA.Interval(-3.0, 1.0)
p1 = Taylor1([2.0, 1.0], 2)
p2 = Taylor1([0.9, 3.0], 2)

# define vector of linear Taylor models
vTMlin = [TM1(pi, I, x₀, D) for pi in [p1,p2]]

# define vector of non-linear Taylor models
p2nl = Taylor1([0.9, 3.0, 3.0], 3)
vTMnonlin = [TM1(pi, I, x₀, D) for pi in [p1, p2nl]]
```

C.5 Code for Section 5.5

Special array operations and the type `SingleEntryVector` are available in the LazySets submodule `LazySets.Arrays` . This example consists of the concrete intersection of two 10-dimensional sets, one of them which is unbounded (a half-space).

```
const SEV = LazySets.Arrays.SingleEntryVector

X = Hyperrectangle(zeros(10), 2*ones(10))
Hsev = HalfSpace(SEV(1, 10, 1.0)), 1.0)
Hvec = HalfSpace(Vector(Hsev.a), 1.0)
```