

---

# **pycvxset Documentation**

***Release 1.2.0***

**Abraham P. Vinod**

***Mitsubishi Electric Research Laboratories (MERL)***

**Email: [vinod@merl.com](mailto:vinod@merl.com), [abraham.p.vinod@ieee.org](mailto:abraham.p.vinod@ieee.org)**

**Feb 13, 2026**

# CONTENTS

<b>1</b>	<b>What is pycvxset?</b>	<b>1</b>
<b>2</b>	<b>Quick start</b>	<b>2</b>
<b>3</b>	<b>Getting help</b>	<b>5</b>
<b>4</b>	<b>Contributing</b>	<b>6</b>
<b>5</b>	<b>License</b>	<b>7</b>
<b>6</b>	<b>Acknowledgements</b>	<b>8</b>
<b>7</b>	<b>Citing</b>	<b>9</b>
<b>8</b>	<b>Contact</b>	<b>10</b>
<b>9</b>	<b>pycvxset.Polytope</b>	<b>11</b>
<b>10</b>	<b>pycvxset.Polytope (API details)</b>	<b>16</b>
<b>11</b>	<b>pycvxset.Ellipsoid</b>	<b>41</b>
<b>12</b>	<b>pycvxset.Ellipsoid (API details)</b>	<b>44</b>
<b>13</b>	<b>pycvxset.ConstrainedZonotope</b>	<b>59</b>
<b>14</b>	<b>pycvxset.ConstrainedZonotope (API details)</b>	<b>63</b>
<b>15</b>	<b>pycvxset.common</b>	<b>83</b>
<b>16</b>	<b>pycvxset.common.constants</b>	<b>89</b>
<b>17</b>	<b>Tutorials</b>	<b>91</b>
<b>18</b>	<b>References</b>	<b>92</b>
	<b>Bibliography</b>	<b>93</b>
	<b>Python Module Index</b>	<b>94</b>
	<b>Index</b>	<b>95</b>

## WHAT IS PYCVXSET?

`pycvxset` (pronounced “Pie CVX Set”) is a Python package for manipulation and visualization of convex sets. The code is open-source (see <https://github.com/merlresearch/pycvxset>), and is maintained by [Abraham P. Vinod](#).

Currently, `pycvxset` supports the following three representations:

1. **polytopes**,
2. **ellipsoids**, and
3. **constrained zonotopes** (which are equivalent to polytopes).

Some of the operations enabled by `pycvxset` include:

- construct sets from a variety of representations and transform between these representations,
- perform various operations on these sets including (but not limited to):
  - plot sets in 2D and 3D,
  - affine and inverse-affine transformation,
  - checking for containment,
  - intersection,
  - Minkowski sum,
  - Pontryagin difference,
  - projection,
  - slicing, and
  - support function evaluation.

See the Jupyter notebooks in [examples](#) folder for more details on how `pycvxset` can be used in set-based control and perform reachability analysis.

## QUICK START

### 2.1 Requirements

pycvxset supports Python 3.9+ on Ubuntu, Windows, and MacOS. As described in [pyproject.toml](#), pycvxset has the following core dependencies:

1. `numpy`
2. `scipy`
3. `cvxpy`
4. `matplotlib`
5. `pycddlib`: pycvxset requires `pycddlib`  $\geq 3.0.0$ . Use pycvxset  $\leq 1.0.2$  to use pycvxset with earlier versions of `pycddlib`  $\leq 2.1.8.post1$ .
6. `gurobipy`: This dependency is **optional**. Almost all functionalities of pycvxset are available without `Gurobi`. However, pycvxset uses `Gurobi` (through `cvxpy`) to perform *some* containment and equality checks involving constrained zonotopes. See [License](#) section for more details.

### 2.2 Installation

Refer to [.github/workflows](#) for exact steps to install pycvxset for different OSes. These steps are summarized below:

1. OS-dependent pre-installation steps:

- **Ubuntu**: Install `gmp`.

```
$ sudo apt-get install libgmp-dev libcdd-dev python3-dev
```

See [pycddlib documentation](#) or [pycddlib/build.yml](#) for more details.

- **MacOS**: Install `gmp` and `cddlib`.

```
% brew install gmp cddlib
% python3 -m pip install --upgrade pip
% env "CFLAGS=-I$(brew --prefix)/include -L$(brew --prefix)/lib" python -m
↳ pip install pycddlib==3.0.0
```

See [pycddlib documentation](#) or [pycddlib/build.yml](#) for more details.

- **Windows**: No special steps required since pip takes care of it. If plotting fails, you can set matplotlib backend via an environment variable set `MPLBACKEND=Agg`. See <https://matplotlib.org/3.5.1/users/explain/backends.html#selecting-a-backend> for more details.

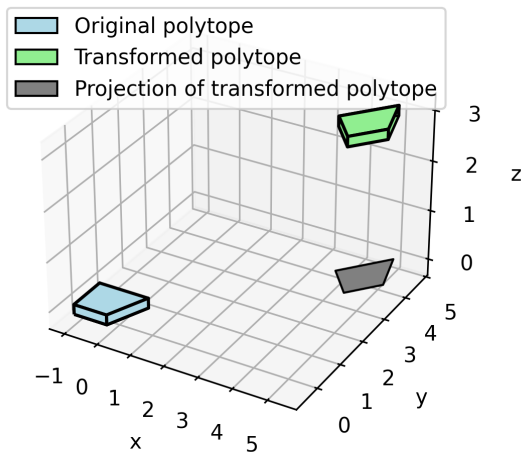
2. Clone the pycvxset repository into a desired folder `PYCVXSET_DIR`.

3. Run `pip install -e .` in the folder `PYCVXSET_DIR`.

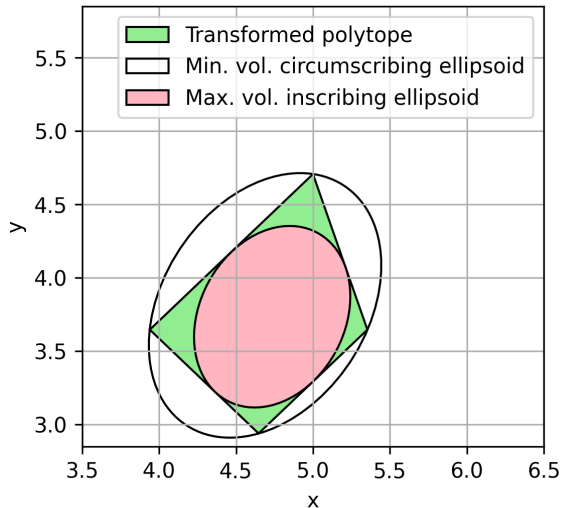
**Sanity check:** Check your installation by running `python3 examples/pycvxset_diag.py` in the folder `PYCVXSET_DIR`. The script should generate a figure with two subplots, each generated using pycvxset.

- *Left subplot:* A 3D plot of two polytopes (one at the origin, and the other translated and rotated). You can interact with this plot using your mouse.
- *Right subplot:* A 2D plot of the projection of the polytope, and its corresponding minimum volume circumscribing and maximum volume inscribing ellipsoids.

Rotate a polytope about z-axis by  $45.0^\circ$   
and shift it by [5, 4, 3]



Projection of transformed polytope on to XY  
and ellipsoidal approximations



The following code block is reproduced from `run_demo` function in `examples/pycvxset_diag.py`. The code block demonstrates how one can use `pycvxset` to define sets and manipulate them:

```
# Copyright (C) 2020-2026 Mitsubishi Electric Research Laboratories (MERL)

import numpy as np
from pycvxset import Ellipsoid, Polytope
from scipy.spatial.transform import Rotation

# Define P as the intersection of a box with different sides and a halfspace
box_with_different_sides = Polytope(c=[0, 0, 0], h=[1, 0.5, 0.1])
P = box_with_different_sides.intersection_with_halfspaces([1, -0.5, 0], 0.25)
# Affine transformation (Rotate and translate P)
rotate_angle = np.pi / 4
R = Rotation.from_rotvec(rotate_angle * np.array([0, 0, 1])).as_matrix()
shift_vec = [5, 4, 3]
transformed_P = R @ P + shift_vec
# Projection (Compute its shadow on to the xy space)
project_transformed_P_to_XY = transformed_P.projection(project_away_dims=2)
# Centering ellipsoids
ellipsoid_inside_projection = Ellipsoid.deflate(project_transformed_P_to_XY)
ellipsoid_outside_projection = Ellipsoid.inflate(project_transformed_P_to_XY)
```

## 2.3 Optional: Testing

1. Use `pip install -e ".[with_tests]"` to install the additional dependencies.
2. Run `$ ./scripts/run_tests_and_update_docs.sh` to view testing results on the command window.
3. View code coverage from testing at `./docs/build/_static/codecoverage/overall/index.html` on your browser.

## 2.4 Optional: Documentation

1. Use `pip install -e "[with_docs_and_tests]"` to install the additional dependencies.
2. To produce the documentation,
  - Run `./scripts/run_all.sh`. This should take about 15 minutes. For **faster but incomplete options**,
    - To generate just the latex files for the [MANUAL.pdf](#), run `./scripts/run_sphinx_pdf.sh`. This command will assume that the environment has [LaTeX](#) setup properly.
    - To build the API documentation without rendering the notebooks or coverage results, run `./scripts/run_sphinx_html.sh`.
3. View the documentation as,
  - PDF at [./MANUAL.pdf](#)
  - HTML pages (without internet) at `localhost:8000` after running `python -m http.server --directory ./docs/build/`.
    - Documentation website is available at `./docs/build/index.html`.
    - View the Jupyter notebooks at `./docs/build/tutorials/tutorials.html`.
    - View code coverage from testing at `./docs/build/_static/codecoverage/overall/index.html`.

## GETTING HELP

See [MANUAL.pdf](#) or <https://merlresearch.github.io/pycvxset/> for documentation of pycvxset.

Please use the [Discussion page](#) or the [Issue page](#), if you still need help.

## CONTRIBUTING

See [CONTRIBUTING.md](#) for our policy on contributions.



## LICENSE

pycvxset code is released under [AGPL-3.0-or-later](#) license, as found in the [LICENSE.md](#) file. The documentation for pycvxset is released under [CC-BY-4.0](#) license, as found in the [LICENSES/CC-BY-4.0.md](#).

All files:

```
Copyright (c) 2020-2026 Mitsubishi Electric Research Laboratories (MERL).
```

```
SPDX-License-Identifier: AGPL-3.0-or-later
```

except the following files:

1. .gitignore
2. README.md
3. pycvxset/Polytope/\_\_init\_\_.py
4. pycvxset/Polytope/operations\_binary.py
5. pycvxset/Polytope/operations\_unary.py
6. pycvxset/Polytope/plotting\_scripts.py
7. pycvxset/Polytope/vertex\_halfspace\_enumeration.py
8. pycvxset/\_\_init\_\_.py
9. pycvxset/common/\_\_init\_\_.py
10. tests/test\_polytope\_binary.py
11. tests/test\_polytope\_init.py
12. tests/test\_polytope\_vertex\_facet\_enum.py

which have the copyright

```
Copyright (C) 2020-2026 Mitsubishi Electric Research Laboratories (MERL)
```

```
Copyright (c) 2019 Tor Aksel N. Heirung
```

```
SPDX-License-Identifier: AGPL-3.0-or-later
```

```
SPDX-License-Identifier: MIT
```

The method contains in [pycvxset/ConstrainedZonotope/operations\\_binary.py](#) uses [gurobipy](#) (through [cvxpy](#)) and requires acceptance of appropriate license terms.

## ACKNOWLEDGEMENTS

The development of `pycvxset` started from commit [ebd85404](#) of `pytope`.

Copyright (c) 2019 Tor Aksel N. Heirung

SPDX-License-Identifier: MIT

`pycvxset` extends `pytope` in several new directions, including:

- Plotting for 3D polytopes using `matplotlib`,
- Interface with `cvxpy` for access to a wider range of solvers,
- Introduce new methods for *Polytope* class including circumscribing and inscribing ellipsoids, volume computation, and containment checks,
- Minimize the use of vertex-halfspace enumeration using convex optimization,
- Include extensive documentation and example Jupyter notebooks,
- Implement exhaustive testing with nearly 100% coverage, and
- Support for *Ellipsoid* and *ConstrainedZonotope* set representations.

## CITING

If you use this software in your research, please cite it using the metadata from [CITATION.cff](#) or the following bibtex entry.

```
@inproceedings{vinod2024pycvxset,  
  title={{pycvxset: A Python package for convex set manipulation}},  
  author={Vinod, Abraham P},  
  booktitle={Proceedings of American Control Conference (ACC)},  
  year={2025},  
  note = {(accepted)}  
}
```

## CONTACT

For questions or bugs, contact Abraham P. Vinod (Email: [vinod@merl.com](mailto:vinod@merl.com), [abraham.p.vinod@ieee.org](mailto:abraham.p.vinod@ieee.org)).

## PYCVXSET.POLYTOPE

**class** pycvxset.Polytope

Bases: object

Polytope class.

Polytope object construction admits **one** of the following combinations (as keyword arguments):

1. (A, b) for a polytope in **halfspace representation (H-rep)**  $\{x \mid Ax \leq b\}$ ,
2. (A, b, Ae, be) for a polytope in **halfspace representation (H-rep) with equality constraints**  $\{x \mid Ax \leq b, Ae x = be\}$ ,
3. V for a polytope in **vertex representation (V-rep)** —  $\text{ConvexHull}(v_i)$  where  $v_i$  are rows of matrix V,
4. (lb, ub) for an **axis-aligned cuboid** with appropriate bounds  $\{x \mid lb \leq x \leq ub\}$ , and
5. (c, h) for an **axis-aligned cuboid** centered at c with specified scalar/vector half-sides h,  $\{x \mid \forall i \in \{1, \dots, n\}, |x_i - c_i| \leq h_i\}$ .
6. dim for an **empty** Polytope of dimension dim (no argument generates a zero-dimensional **empty** Polytope),

#### Parameters

- **dim** (*int*, *optional*) – Dimension of the empty polytope. If NOTHING is provided, dim=0 is assumed.
- **V** (*Sequence[Sequence[float]]* | *np.ndarray*, *optional*) – List of vertices of the polytope (V-Rep). The list must be 2-dimensional with vertices arranged row-wise and the polytope dimension determined by the column count.
- **A** (*Sequence[Sequence[float]]* | *np.ndarray*, *optional*) – Inequality coefficient vectors (H-Rep). The vectors are stacked vertically with the polytope dimension determined by the column count. When A is provided, b must also be provided.
- **b** (*Sequence[float]* | *np.ndarray*, *optional*) – Inequality constants (H-Rep). The constants are expected to be in a 1D numpy array. When b is provided, A must also be provided.
- **Ae** (*Sequence[Sequence[float]]* | *np.ndarray*) – Equality coefficient vectors (H-Rep). The vectors are stacked vertically with matching number of columns as A. When Ae is provided, A, b, and be must also be provided.
- **be** (*Sequence[float]* | *np.ndarray*) – Equality coefficient constants (H-Rep). The constants are expected to be in a 1D numpy array. When be is provided, A, b, and Ae must also be provided.
- **lb** (*Sequence[float]* | *np.ndarray*, *optional*) – Lower bounds of the axis-aligned cuboid. Must be 1D array, and the polytope dimension is determined by number of elements in lb. When lb is provided, ub must also be provided.

- **ub** (*Sequence[float] | np.ndarray, optional*) – Upper bounds of the axis-aligned cuboid. Must be 1D array of length as same as lb. When ub is provided, lb must also be provided.
- **c** (*Sequence[float] | np.ndarray, optional*) – Center of the axis-aligned cuboid. Must be 1D array, and the polytope dimension is determined by number of elements in c. When c is provided, h must also be provided.
- **h** (*float | Sequence[float] | np.ndarray, optional*) – Half-side length of the axis-aligned cuboid. Can be a scalar or a vector of length as same as c. When h is provided, c must also be provided.

#### Raises

- **ValueError** – When arguments provided is not one of [(A, b), (A, b, Ae, be), (lb, ub), (c, h), V, dim, NOTHING]
- **ValueError** – Errors raised by issues with (A, b) and (Ae, be) — mismatch in dimensions, not convertible to appropriately-dimensioned numpy arrays, incompatible systems (A, b) and (Ae, be) etc.
- **ValueError** – Errors raised by issues with V — not convertible to a 2-D numpy array, etc.
- **ValueError** – Errors raised by issues with lb, ub — mismatch in dimensions, not convertible to 1D numpy arrays, etc.
- **ValueError** – Errors raised by issues with c, h — mismatch in dimensions, not convertible to 1D numpy arrays, etc.
- **ValueError** – Polytope is not bounded in any direction. We use a sufficient condition for speed, and therefore the detection may not be exhaustive.
- **ValueError** – Polytope is not bounded in some directions. We use a sufficient condition for speed, and therefore the detection may not be exhaustive.
- **UserWarning** – If some rows are removed from (A, b) due to all zeros in A or np.inf in b | (Ae, be) when all zeros in Ae and be.

`__init__()` → None

`__init__(*, dim: int)` → None

`__init__(*, V: Sequence[Sequence[float]] | ndarray)` → None

`__init__(*, A: Sequence[Sequence[float]] | ndarray, b: Sequence[float] | ndarray)` → None

`__init__(*, A: Sequence[Sequence[float]] | ndarray, b: Sequence[float] | ndarray, Ae: Sequence[Sequence[float]] | ndarray, be: Sequence[float] | ndarray)` → None

`__init__(*, lb: Sequence[float] | ndarray, ub: Sequence[float] | ndarray)` → None

`__init__(*, c: Sequence[float] | ndarray, h: float | Sequence[float] | ndarray)` → None

Constructor for Polytope class.

#### Methods

<code>__init__()</code>	Constructor for Polytope class.
<code>affine_map(M)</code>	Compute the matrix times set.
<code>chebyshev_centering()</code>	Computes a ball with the largest radius that fits within the polytope.
<code>closest_point(points[, p])</code>	Wrapper for <code>project()</code> to compute the point in the convex set closest to the given point.
<code>containment_constraints(x[, flatten_order])</code>	Get CVXPY constraints for containment of x (a <code>cvxpy.Variable</code> ) in a polytope.

continues on next page

Table 1 – continued from previous page

<code>contains(Q)</code>	Check containment of a set $Q$ (could be a polytope, an ellipsoid, or a constrained zonotope), or a collection of points $Q \in \mathbb{R}^{n_Q \times \mathcal{P}.dim}$ in the polytope $\mathcal{P}$ .
<code>copy()</code>	Create a copy of the polytope
<code>decompose_as_affine_transform_of_polytope</code>	Express a polytope with equality constraints as an affine transformation of a lower-dimensional polytope without any equality constraints (hence full-dimensional).
<code>deflate_rectangle(set_to_be_centered)</code>	Compute the rectangle with the smallest volume that contains the given set.
<code>determine_H_rep([prefer_qhull_over_cdd, prune_V])</code>	Determine the halfspace representation from a given vertex representation of the polytope.
<code>determine_V_rep([prefer_qhull_over_cdd])</code>	Determine the vertex representation from a given halfspace representation of the polytope.
<code>distance(points[, p])</code>	Wrapper for <code>project()</code> to compute distance of a point to a convex set.
<code>extreme(eta)</code>	Wrapper for <code>support()</code> to compute the extreme point.
<code>interior_point([point_type])</code>	Compute a point in the interior of the polytope.
<code>intersection(Q)</code>	Intersect the polytope $\mathcal{P}$ with another polytope $Q$ .
<code>intersection_under_inverse_affine_map(Q, R)</code>	Compute the intersection of constrained zonotope under an inverse affine map
<code>intersection_with_affine_set(Ae, be)</code>	Intersect the polytope with an affine set.
<code>intersection_with_halfspaces(A, b)</code>	Intersect the polytope with a collection of halfspaces.
<code>inverse_affine_map_under_invertible_matrix</code>	Compute the set times matrix, the inverse affine map of the set under an invertible matrix $M$ .
<code>maximum_volume_inscribing_ellipsoid()</code>	Compute the maximum volume ellipsoid that fits within the given polytope.
<code>minimize(x, objective_to_minimize, cvxpy_args)</code>	Solve a convex program with CVXPY objective subject to containment constraints.
<code>minimize_H_rep()</code>	Remove any redundant inequalities from the halfspace representation of the polytope using cdd.
<code>minimize_V_rep([prefer_qhull_over_cdd, prune_V])</code>	Remove any redundant vertices from the vertex representation of the polytope.
<code>minimum_volume_circumscribing_ellipsoid()</code>	Compute the minimum volume ellipsoid that covers the given polytope (also known as Lowner-John Ellipsoid).
<code>minimum_volume_circumscribing_rectangle()</code>	Compute the minimum volume circumscribing rectangle for a given polytope
<code>minus(Q)</code>	Implement - operation (Pontryagin difference when $Q$ is a polytope, translation by $-Q$ when $Q$ is a point)
<code>normalize()</code>	Normalize a H-Rep such that each row of $A$ has unit $\ell_2$ -norm.
<code>plot([ax, patch_args, vertex_args, ...])</code>	Plot a 2D or 3D polytope.
<code>plot2d([ax, patch_args, vertex_args, ...])</code>	Plot a 2D polytope using matplotlib's <code>add_patch(Polygon())</code> .
<code>plot3d([ax, patch_args, vertex_args, ...])</code>	Plot a 3D polytope using matplotlib's <code>Line3DCollection</code> .
<code>plus(Q)</code>	Add a point or a set to the polytope
<code>project(x[, p])</code>	Project a point or a collection of points on to a set.
<code>projection(project_away_dims)</code>	Orthogonal projection of a set $\mathcal{P}$ after removing some user-specified dimensions.
<code>slice(dims, constants)</code>	Slice a set restricting certain dimensions to constants.
<code>slice_then_projection(dims, constants)</code>	Wrapper for <code>slice()</code> and <code>projection()</code> .

continues on next page

Table 1 – continued from previous page

<code>support(eta)</code>	Evaluates the support function and support vector of a set.
<code>volume()</code>	Compute the volume of the polytope using QHull

**Attributes**

<code>A</code>	Inequality coefficient vectors $A$ for the polytope $\{Ax \leq b, A_e x = b_e\}$ .
<code>Ae</code>	Equality coefficient vectors $A_e$ for the polytope $\{Ax \leq b, A_e x = b_e\}$ .
<code>H</code>	Inequality constraints in halfspace representation $H=[A, b]$ for the polytope $\{Ax \leq b, A_e x = b_e\}$ .
<code>He</code>	Equality constraints in halfspace representation $He=[A_e, b_e]$ for the polytope $\{Ax \leq b, A_e x = b_e\}$ .
<code>V</code>	Vertex representation ( $V$ ) where the polytope is given by $\text{ConvexHull}(v_i)$ with $v_i$ as the rows of $V = [v_1; v_2; \dots; v_{n_{\text{vertices}}}]$ .
<code>b</code>	Inequality constants $b$ for the polytope $\{Ax \leq b, A_e x = b_e\}$ .
<code>be</code>	Equality constants $b_e$ for the polytope $\{Ax \leq b, A_e x = b_e\}$ .
<code>cvxpy_args_lp</code>	CVXPY arguments in use when solving a linear program
<code>cvxpy_args_sdp</code>	CVXPY arguments in use when solving a semi-definite program
<code>cvxpy_args_socp</code>	CVXPY arguments in use when solving a second-order cone program
<code>dim</code>	Dimension of the polytope.
<code>in_H_rep</code>	Check if the polytope have a halfspace representation (H-Rep).
<code>in_V_rep</code>	Check if the polytope have a vertex representation (V-Rep).
<code>is_bounded</code>	Check if the polytope is bounded.
<code>is_empty</code>	Check if the polytope is empty.
<code>is_full_dimensional</code>	Check if the affine dimension of the polytope is the same as the polytope dimension
<code>is_singleton</code>	Check if the polytope is singleton.
<code>n_equalities</code>	Number of linear equality constraints used to define the polytope $\{Ax \leq b, A_e x = b_e\}$
<code>n_halfspaces</code>	Number of halfspaces used to define the polytope $\{Ax \leq b, A_e x = b_e\}$
<code>n_vertices</code>	Number of vertices
<code>type_of_set</code>	Return the type of set

`__init__()` → None`__init__(*, dim: int)` → None`__init__(*, V: Sequence[Sequence[float]] | ndarray)` → None`__init__(*, A: Sequence[Sequence[float]] | ndarray, b: Sequence[float] | ndarray)` → None`__init__(*, A: Sequence[Sequence[float]] | ndarray, b: Sequence[float] | ndarray, Ae: Sequence[Sequence[float]] | ndarray, be: Sequence[float] | ndarray)` → None`__init__(*, lb: Sequence[float] | ndarray, ub: Sequence[float] | ndarray)` → None`__init__(*, c: Sequence[float] | ndarray, h: float | Sequence[float] | ndarray)` → None

Constructor for Polytope class.



`__new__`(\*\**kwargs*)

## PYCVXSET.POLYTOPE (API DETAILS)

`pycvxset.Polytope`

Polytope class.

**class** pycvxset.Polytope.Polytope

Bases: object

Polytope class.

Polytope object construction admits **one** of the following combinations (as keyword arguments):

1. (A, b) for a polytope in **halfspace representation (H-rep)**  $\{x \mid Ax \leq b\}$ ,
2. (A, b, Ae, be) for a polytope in **halfspace representation (H-rep) with equality constraints**  $\{x \mid Ax \leq b, A_e x = b_e\}$ ,
3. V for a polytope in **vertex representation (V-rep)** —  $\text{ConvexHull}(v_i)$  where  $v_i$  are rows of matrix V,
4. (lb, ub) for an **axis-aligned cuboid** with appropriate bounds  $\{x \mid lb \leq x \leq ub\}$ , and
5. (c, h) for an **axis-aligned cuboid** centered at c with specified scalar/vector half-sides h,  $\{x \mid \forall i \in \{1, \dots, n\}, |x_i - c_i| \leq h_i\}$ .
6. dim for an **empty** Polytope of dimension dim (no argument generates a zero-dimensional **empty** Polytope),

**Parameters**

- **dim** (*int, optional*) – Dimension of the empty polytope. If NOTHING is provided, dim=0 is assumed.
- **V** (*Sequence[Sequence[float]] | np.ndarray, optional*) – List of vertices of the polytope (V-Rep). The list must be 2-dimensional with vertices arranged row-wise and the polytope dimension determined by the column count.
- **A** (*Sequence[Sequence[float]] | np.ndarray, optional*) – Inequality coefficient vectors (H-Rep). The vectors are stacked vertically with the polytope dimension determined by the column count. When A is provided, b must also be provided.
- **b** (*Sequence[float] | np.ndarray, optional*) – Inequality constants (H-Rep). The constants are expected to be in a 1D numpy array. When b is provided, A must also be provided.
- **Ae** (*Sequence[Sequence[float]] | np.ndarray*) – Equality coefficient vectors (H-Rep). The vectors are stacked vertically with matching number of columns as A. When Ae is provided, A, b, and be must also be provided.
- **be** (*Sequence[float] | np.ndarray*) – Equality coefficient constants (H-Rep). The constants are expected to be in a 1D numpy array. When be is provided, A, b, and Ae must also be provided.

- **lb** (*Sequence[float] | np.ndarray, optional*) – Lower bounds of the axis-aligned cuboid. Must be 1D array, and the polytope dimension is determined by number of elements in lb. When lb is provided, ub must also be provided.
- **ub** (*Sequence[float] | np.ndarray, optional*) – Upper bounds of the axis-aligned cuboid. Must be 1D array of length as same as lb. When ub is provided, lb must also be provided.
- **c** (*Sequence[float] | np.ndarray, optional*) – Center of the axis-aligned cuboid. Must be 1D array, and the polytope dimension is determined by number of elements in c. When c is provided, h must also be provided.
- **h** (*float | Sequence[float] | np.ndarray, optional*) – Half-side length of the axis-aligned cuboid. Can be a scalar or a vector of length as same as c. When h is provided, c must also be provided.

#### Raises

- **ValueError** – When arguments provided is not one of [(A, b), (A, b, Ae, be), (lb, ub), (c, h), V, dim, NOTHING]
- **ValueError** – Errors raised by issues with (A, b) and (Ae, be) — mismatch in dimensions, not convertible to appropriately-dimensioned numpy arrays, incompatible systems (A, b) and (Ae, be) etc.
- **ValueError** – Errors raised by issues with V — not convertible to a 2-D numpy array, etc.
- **ValueError** – Errors raised by issues with lb, ub — mismatch in dimensions, not convertible to 1D numpy arrays, etc.
- **ValueError** – Errors raised by issues with c, h — mismatch in dimensions, not convertible to 1D numpy arrays, etc.
- **ValueError** – Polytope is not bounded in any direction. We use a sufficient condition for speed, and therefore the detection may not be exhaustive.
- **ValueError** – Polytope is not bounded in some directions. We use a sufficient condition for speed, and therefore the detection may not be exhaustive.
- **UserWarning** – If some rows are removed from (A, b) due to all zeros in A or np.inf in b | (Ae, be) when all zeros in Ae and be.

`__init__()` → None

`__init__(*, dim: int)` → None

`__init__(*, V: Sequence[Sequence[float]] | ndarray)` → None

`__init__(*, A: Sequence[Sequence[float]] | ndarray, b: Sequence[float] | ndarray)` → None

`__init__(*, A: Sequence[Sequence[float]] | ndarray, b: Sequence[float] | ndarray, Ae: Sequence[Sequence[float]] | ndarray, be: Sequence[float] | ndarray)` → None

`__init__(*, lb: Sequence[float] | ndarray, ub: Sequence[float] | ndarray)` → None

`__init__(*, c: Sequence[float] | ndarray, h: float | Sequence[float] | ndarray)` → None

Constructor for Polytope class.

`affine_map(M: Sequence[float] | Sequence[Sequence[float]] | np.ndarray)` → *Polytope*

Compute the matrix times set.

#### Parameters

**M** (*Sequence[float] | Sequence[Sequence[float]] | np.ndarray*) – A vector or array (0, 1, or 2-D) with self.dim columns

#### Raises

- **ValueError** – When M can not be converted EXACTLY into a 2D array of float
- **ValueError** – When M does not have self.dim columns

### Returns

The scaled polytope  $\mathcal{R} = M\mathcal{P} = \{Mx : x \in \mathcal{P}\}$

### Return type

*Polytope*

### Notes

This function requires  $\mathcal{P}$  to be in V-Rep, and performs a vertex enumeration when  $\mathcal{P}$  is in H-Rep.

**chebyshev\_centering()**  $\rightarrow$  tuple[np.ndarray | None, float]

Computes a ball with the largest radius that fits within the polytope. The ball's center is known as the Chebyshev center, and its radius is the Chebyshev radius.

### Raises

**NotImplementedError** – Unable to solve chebyshev centering problem using CVXPY

### Returns

**A tuple with two items**

1. center (numpy.ndarray): Chebyshev center of the polytope
2. radius (float): Chebyshev radius of the polytope

### Return type

tuple

### Notes

This function is called by the constructor for non-emptiness and boundedness check. It uses limited attributes (dim, A, b, Ae, be). This function requires H-Rep, and will perform a halfspace enumeration when a V-Rep polytope is provided.

We solve the LP (see Section 8.5.1 in [BV04]) for  $c$  (for *center*) and  $R$  (for *radius*),

$$\begin{aligned} &\text{maximize} && R \\ &\text{subject to} && Ac + R\|A\|_{\text{row}} \leq b, \\ & && A_e c = b_e, \\ & && R \geq 0, \end{aligned}$$

where  $\|A\|_{\text{row}}$  is a vector of dimension `n_halfspaces` with each element as  $\|a_i\|_2$ . When (Ae, be) is non-empty, then R is forced to zero post-solve. Consequently, chebyshev\_centering also serves a method to find a feasible point in the relative interior of the polytope.

**We can also infer the following from the Chebyshev radius R:**

1.  $R = \infty$ , the polytope is unbounded. Note that, this is just a sufficient condition, and an unbounded polytope can have a finite Chebyshev radius. For example, consider a 3-dimensional axis-aligned cuboid  $[-1, 1] \times [-1, 1] \times \mathbb{R}$ .
2.  $0 < R < \infty$ , the polytope is nonempty and full-dimensional.
3.  $R = 0$ , the polytope is nonempty and but not full-dimensional.
4.  $R = -\infty$ , the polytope is empty.

**closest\_point(points: Sequence[Sequence[float]] | np.ndarray, p: int | str = 2)  $\rightarrow$  np.ndarray**

Wrapper for [project\(\)](#) to compute the point in the convex set closest to the given point.

### Parameters

- **points** (*Sequence[Sequence[float]]* | *np.ndarray*) – Points to project. Matrix (N times self.dim), where each row is a point.
- **p** (*str* | *int*) – Norm-type. It can be 1, 2, or 'inf'. Defaults to 2.

### Returns

Projection of points to the set as a 2D numpy.ndarray. These arrays have as many rows as points.

### Return type

numpy.ndarray

### Notes

For more detailed description, see documentation for [project\(\)](#) function.

**containment\_constraints**(*x*: cvxpy.Variable, *flatten\_order*: Literal['F', 'C'] = 'F') → tuple[list[cvxpy.Constraint], cvxpy.Variable | None]

Get CVXPY constraints for containment of *x* (a cvxpy.Variable) in a polytope.

### Parameters

- **x** (cvxpy.Variable) – CVXPY variable to be optimized
- **flatten\_order** (Literal["F", "C"]) – Order to use for flatten (choose between “F”, “C”). Defaults to “F”, which implements column-major flatten. In 2D, column-major flatten results in stacking rows horizontally to achieve a single horizontal row.

### Raises

**ValueError** – When polytope is empty

### Returns

A tuple with two items:

1. *constraint\_list* (list): CVXPY constraints for the containment of *x* in the polytope.
2. *theta* (cvxpy.Variable | None): CVXPY variable representing the convex combination coefficient when polytope is in V-Rep. It is None when the polytope is in H-Rep or empty.

### Return type

tuple

**contains**(*Q*: Sequence[float] | Sequence[Sequence[float]] | np.ndarray | ConstrainedZonotope | Ellipsoid | Polytope) → bool | np.ndarray

Check containment of a set *Q* (could be a polytope, an ellipsoid, or a constrained zonotope), or a collection of points  $Q \in \mathbb{R}^{n_Q \times \mathcal{P}.dim}$  in the polytope *P*.

### Parameters

**Q** (Sequence[float] | Sequence[Sequence[float]] | np.ndarray | ConstrainedZonotope | Ellipsoid | Polytope) – Set or a collection of points (each row is a point) to be tested for containment within *P*. When providing a collection of points, *Q* is a matrix (N times self.dim) with each row is a point.

### Raises

**ValueError** – When two polytopes | the polytope and the test point(s) are NOT of the same dimension

### Returns

**When *Q* is a polytope, a bool is returned which is True if and only if**

$Q \subseteq P$ . On the other hand, if *Q* is a sequence or numpy array (*Q* is a point or a collection of points), then a numpy.ndarray of bool is returned, with as many elements as the number of rows in *Q*.

### Return type

bool | numpy.ndarray[bool]

## Notes

- *Containment of polytopes*: This function accommodates the following combinations of representations for  $\mathcal{P}$  and  $\mathcal{Q}$ . It eliminates the need for enumeration by comparing support function evaluations almost always. When  $\mathcal{P}$  is H-Rep and  $\mathcal{Q}$  is V-Rep, then we perform a quick check for containment of vertices of  $\mathcal{Q}$  in  $\mathcal{P}$ . Otherwise, \*  $\mathcal{Q}$  is empty: Return True \*  $\mathcal{P}$  is empty: Return False
- *Containment of points*: This function accommodates  $\mathcal{P}$  to be in H-Rep or V-Rep. When testing if a point is in a polytope where only V-Rep is available for self, we solve a collection of second-order cone programs for each point using `project()` (check if distance between  $v$  and  $\mathcal{P}$  is nearly zero). Otherwise, we use the observation that for  $\mathcal{P}$  with  $(A, b, A_e, b_e)$ , then  $v \in \mathcal{P}$  if and only if  $Av \leq b$  and  $A_e v = b_e$ . For numerical precision considerations, we use `numpy.isclose()`.

`copy()` → *Polytope*

Create a copy of the polytope

`decompose_as_affine_transform_of_polytope_without_equalities()` → tuple[*Polytope*,  
np.ndarray, np.ndarray]

Express a polytope with equality constraints as an affine transformation of a lower-dimensional polytope without any equality constraints (hence full-dimensional). The affine transformation is obtained via a QR decomposition.

If the polytope is already full-dimensional, returns a copy of itself and the identity matrix. Otherwise, finds an affine transformation that maps a lower-but-full-dimensional polytope to the original polytope.

### Returns

**A tuple with three items:**

1. `full_dimensional_polytope` (*Polytope*): A full-dimensional polytope.
2. `affine_transform_offset` (numpy.ndarray): The affine offset (as a vector) to map the full-dimensional polytope to the original polytope.
3. `affine_transform_matrix` (numpy.ndarray): A matrix whose columns form an orthonormal basis for the nullspace of the affine equality constraints.

### Return type

tuple

## Notes

This function requires H-Rep, and will perform a halfspace enumeration when a V-Rep polytope is provided. The decomposition is such that  $self = M @ full\_dim\_polytope\_in\_relative\_interior\_of\_self + c$ . The function uses QR decomposition to find  $M$  and  $c$ . To avoid numerical issues, this function calls `pycvxset.Polytope.minimize_H_rep()`.

`classmethod deflate_rectangle`(*set\_to\_be\_centered*: *ConstrainedZonotope* | *Ellipsoid* | *Polytope*)  
→ *Polytope*

Compute the rectangle with the smallest volume that contains the given set.

### Parameters

**set\_to\_be\_centered** (*ConstrainedZonotope* | *Ellipsoid* | *Polytope*) – Set to compute the.

### Returns

Minimum volume circumscribing rectangle

### Return type

*Polytope*

## Notes

This function is a wrapper for `minimum_volume_circumscribing_rectangle()` of `attr:set_to_be_centered`. Please check that function for more details including raising exceptions.

**determine\_H\_rep**(*prefer\_qhull\_over\_cdd*: bool = False, *prune\_V*: bool = False) → None

Determine the halfspace representation from a given vertex representation of the polytope.

### Parameters

- **prefer\_qhull\_over\_cdd** (bool, optional) – When True, determine\_H\_rep uses qhull when possible. Otherwise, we use cdd. Default is True.
- **prune\_V** (bool, optional) – When True, prune vertices before halfspace enumeration. Defaults to False.

### Raises

**ValueError** – When H-rep computation fails OR Polytope is not bounded!

## Notes

- When the set is empty, we define an empty polytope.
- Otherwise, we use cdd for the halfspace enumeration.
- The computed vertex representation need not be minimal.
- We do not check for bounded polytope since we have V-Rep.

**determine\_V\_rep**(*prefer\_qhull\_over\_cdd*: bool = False) → None

Determine the vertex representation from a given halfspace representation of the polytope.

### Parameters

**prefer\_qhull\_over\_cdd** (bool, optional) – When True, determine\_V\_rep uses qhull when possible. Otherwise, we use cdd. Default is True.

### Raises

- **ValueError** – Vertex enumeration yielded rays! Possibly due to numerical issues or unbounded polytope!
- **ValueError** – Polytope is not bounded!

## Notes

We use cdd for the vertex enumeration. cdd uses the halfspace representation  $[b, -A]$  where  $b - Ax \geq 0 \Leftrightarrow Ax \leq b$ .

For a polyhedron described as

$$\mathcal{P} = \text{conv}(v_1, \dots, v_n) + \text{nonneg}(r_1, \dots, r_s),$$

the V-representation matrix in cdd is  $[t \ V]$  where  $t$  is the column vector with  $n$  ones followed by  $s$  zeroes, and  $V$  is the stacked matrix of  $n$  vertex row vectors on top of  $s$  ray row vectors. `pycvxset` uses only bounded polyhedron, so we should never observe rays.

**distance**(*points*: Sequence[Sequence[float]] | np.ndarray, *p*: int | str = 2) → np.ndarray

Wrapper for `project()` to compute distance of a point to a convex set.

### Parameters

- **points** (Sequence[Sequence[float]] | np.ndarray) – Points to project. Matrix ( $N$  times self.dim), where each row is a point.
- **p** (int | str) – Norm-type. It can be 1, 2, or 'inf'. Defaults to 2.

**Returns**

Distance of points to the set as a 1D numpy.ndarray. These arrays have as many rows as points.

**Return type**

numpy.ndarray

**Notes**

For more detailed description, see documentation for [project\(\)](#) function.

**extreme**(*eta*: Sequence[Sequence[float]] | np.ndarray) → np.ndarray

Wrapper for [support\(\)](#) to compute the extreme point.

**Parameters**

**eta** (Sequence[Sequence[float]] | np.ndarray) – Support directions. Matrix (N times self.dim), where each row is a support direction.

**Returns**

Support vector evaluation(s) as a 2D numpy.ndarray. The array has as many rows as eta.

**Return type**

numpy.ndarray

**Notes**

For more detailed description, see documentation for [support\(\)](#) function.

**interior\_point**(*point\_type*: str | None = None) → np.ndarray

Compute a point in the interior of the polytope. When the polytope is not full-dimensional, the point may lie on the boundary.

**Parameters**

**point\_type** (str, optional) – Type of interior point. Valid strings: {'centroid', 'chebyshev', 'mvie'}. Defaults to 'centroid' if has V-Rep and 'chebyshev' if has H-Rep.

**Raises**

- **NotImplementedError** – When an invalid point\_type is provided.
- **ValueError** – When the polytope provided is empty.
- **ValueError** – When the polytope provided is not bounded.

**Returns**

A feasible point in the polytope in the interior as a 1D array

**Return type**

numpy.ndarray

**Notes**

- point\_type is 'centroid': Computes the average of the vertices. The function requires the polytope to be in V-Rep, and a vertex enumeration is performed if the polytope is in H-Rep.
- point\_type is 'chebyshev': Computes the Chebyshev center. The function requires the polytope to be in H-Rep, and a halfspace enumeration is performed if the polytope is in V-Rep.

**intersection**(*Q*: Polytope) → Polytope

Intersect the polytope  $\mathcal{P}$  with another polytope  $\mathcal{Q}$ .

**Parameters**

**Q** (Polytope) – Polytope to be intersected with self

**Raises**

**ValueError** – Q is not a polytope | Mismatch in dimensions



### Returns

The intersection of  $P$  and  $Q$

### Return type

*Polytope*

### Notes

This function requires  $\mathcal{P}$  and  $Q$  to be of the same dimension and in H-Rep, and performs halfspace enumerations when  $P$  or  $Q$  are in V-Rep.

**intersection\_under\_inverse\_affine\_map**( $Q$ : *Polytope*,  $R$ : *Sequence[Sequence[float]]* | *np.ndarray*) → *Polytope*

Compute the intersection of constrained zonotope under an inverse affine map

### Parameters

- **Q** (*Polytope*) – Set to intersect with
- **R** (*Sequence[Sequence[float]]* | *np.ndarray*) – Matrix of dimension  $Y.\text{dim}$  times  $\text{self.dim}$

### Raises

- **ValueError** – When  $Q$  is not a *Polytope*
- **ValueError** – When  $R$  is not of correct dimension
- **ValueError** – When  $\text{self}$  is not bounded

### Returns

The intersection of a polytope with another polytope under an inverse affine map. Specifically, given polytopes  $\mathcal{P}$  ( $\text{self}$ ) and  $Q$ , and a matrix  $R$ , we compute the set  $\{x \in \mathcal{P} \mid Rx \in Q\}$ .

### Return type

*Polytope*

### Notes

This function requires both polytopes to be in H-Rep. Halfspace enumeration is performed when necessary.

Unlike *inverse\_affine\_map\_under\_invertible\_matrix()*, this function does not require  $R$  to be invertible or square and also accommodates  $Q$  to be an unbounded polytope. However,  $\text{self}$  must be a bounded set.

**intersection\_with\_affine\_set**( $A_e$ : *Sequence[Sequence[float]]* | *np.ndarray*,  $b_e$ : *Sequence[float]* | *np.ndarray*) → *Polytope*

Intersect the polytope with an affine set.

### Parameters

- **Ae** (*Sequence[Sequence[float]]* | *np.ndarray*) – Equality coefficient vectors (H-Rep). The vectors are stacked vertically.
- **be** (*Sequence[float]* | *np.ndarray*) – Equality constants (H-Rep). The constants are expected to be in a 1D numpy array.

### Raises

**ValueError** – Mismatch in dimensions | ( $A_e$ ,  $b_e$ ) is not a valid collection of equality constraints.

### Returns

The intersection of  $P$  and  $\{x : A_e x = b_e\}$

### Return type

*Polytope*

## Notes

This function requires  $\mathcal{P}.\text{dim} = A_e.\text{shape}[1]$  and  $\mathcal{P}$  should be in H-Rep, and performs halfspace enumeration if  $\mathcal{P}$  in V-Rep.

**intersection\_with\_halfspaces**( $A$ : *Sequence[Sequence[float]]* | *np.ndarray*,  $b$ : *Sequence[float]* | *np.ndarray*) → *Polytope*

Intersect the polytope with a collection of halfspaces.

### Parameters

- **A** (*Sequence[Sequence[float]]* | *np.ndarray*) – Inequality coefficient vectors (H-Rep). The vectors are stacked vertically.
- **b** (*Sequence[float]* | *np.ndarray*) – Inequality constants (H-Rep). The constants are expected to be in a 1D numpy array.

### Raises

**ValueError** – Mismatch in dimensions | ( $A$ ,  $b$ ) is not a valid collection of halfspaces

### Returns

The intersection of  $P$  and  $\{x \in \mathbb{R}^n \mid Ax \leq b\}$

### Return type

*Polytope*

## Notes

This function requires  $\mathcal{P}.\text{dim} = A.\text{shape}[1]$  and  $\mathcal{P}$  should be in H-Rep, and performs halfspace enumeration if  $\mathcal{P}$  in V-Rep.

**inverse\_affine\_map\_under\_invertible\_matrix**( $M$ : *Sequence[Sequence[float]]* | *np.ndarray*) → *Polytope*

Compute the set times matrix, the inverse affine map of the set under an invertible matrix  $M$ .

### Parameters

**M** (*Sequence[Sequence[float]]* | *np.ndarray*) – An invertible array of size  $\text{self.dim}$  times  $\text{self.dim}$

### Raises

- **ValueError** – When  $\text{self}$  is empty
- **TypeError** – When  $M$  is not convertible into a 2D numpy array of float
- **ValueError** – When  $M$  is not a square matrix
- **ValueError** – When  $M$  is not invertible

### Returns

The inverse-scaled polytope  $\mathcal{R} = \mathcal{P} \times M = \{x : Mx \in \mathcal{P}\}$

### Return type

*Polytope*

## Notes

- This function accommodates  $\mathcal{P}$  to be in H-Rep or in V-Rep. When  $\mathcal{P}$  is in H-Rep,  $\mathcal{P}M = \{x \mid Mx \in \mathcal{P}\} = \{x \mid AMx \leq b, A_e Mx = b_e\}$ . On the other hand, when  $\mathcal{P}$  is in V-Rep,  $\mathcal{P}M = \text{ConvexHull}(M^{-1}v_i)$ .
- We require  $M$  to be invertible in order to ensure that the resulting set is representable as a polytope.

**maximum\_volume\_inscribing\_ellipsoid**() → *tuple*[*np.ndarray*, *np.ndarray*, *np.ndarray*]

Compute the maximum volume ellipsoid that fits within the given polytope.

### Raises

- **ValueError** – When polytope is empty or has non-empty (Ae, be)
- **NotImplementedError** – Unable to solve convex problem using CVXPY

#### Returns

A tuple with three items:

1. center (numpy.ndarray): Maximum volume inscribed ellipsoid's center
2. shape\_matrix (numpy.ndarray): Maximum volume inscribed ellipsoid's shape matrix.
3. sqrt\_shape\_matrix (numpy.ndarray): Maximum volume inscribed ellipsoid's square root of shape matrix. Returns np.zeros((self.dim, 0)) if the polytope is a singleton.

**Return type**  
tuple

#### Notes

This function requires H-Rep, and will perform a vertex enumeration when a V-Rep polytope is provided.

When the polytope is full-dimensional, we can solve a second-order cone program (SOCP). Consider the full-dimensional ellipsoid  $\{Gu + c \mid \|u\|_2 \leq 1\}$ , where  $G$  is a square, lower-triangular matrix with positive diagonal entries. Then, we solve the following (equivalent) optimization problem:

$$\begin{aligned} & \text{minimize} && \text{geomean}(G) \\ & \text{subject to} && \text{diag}(G) \geq 0 \\ & && \|G^T a_i\|_2 + a_i^T d \leq b_i, \end{aligned}$$

with decision variables  $G$  and  $c$ . Here, we use the observation that  $\text{geomean}(G)$  is a monotone function of  $\log \det(GG^T)$  (which is proportional to the volume of the ellipsoid).

When the polytope is not full-dimensional, we first compute the relative interior and then solve the SDP for a positive definite matrix  $B \in \mathbb{S}_{++}^n$  and  $d \in \mathbb{R}^n$ ,

$$\begin{aligned} & \text{maximize} && \log \det B \\ & \text{subject to} && \|Ba_i\|_2 + a_i^T d \leq b_i, \\ & && A_e d = b_e, \\ & && BA_e^T = 0, \end{aligned}$$

where  $(a_i, b_i)$  is the set of hyperplanes characterizing  $\mathcal{P}$ , and the inscribing ellipsoid is given by  $\{Bu + d \mid \|u\|_2 \leq 1\}$ . The center of the ellipsoid is given by  $c = d$ , and the shape matrix is given by  $Q = (BB)^{-1}$ . See [EllipsoidalTbx-Min\_verticesolEll] and Section 8.4.2 in [BV04] for more details. The last two constraints arise from requiring the inscribing ellipsoid to lie in the affine set  $\{A_e x = b_e\}$ .

**minimize**(x: cvxpy.Variable, objective\_to\_minimize: cvxpy.Expression, cvxpy\_args: dict[str, Any], task\_str: str = "") → tuple[np.ndarray, float, str]

Solve a convex program with CVXPY objective subject to containment constraints.

#### Parameters

- **x** (cvxpy.Variable) – CVXPY variable to be optimized
- **objective\_to\_minimize** (cvxpy.Expression) – CVXPY expression to be minimized
- **cvxpy\_args** (dict) – CVXPY arguments to be passed to the solver
- **task\_str** (str, optional) – Task string to be used in error messages. Defaults to “.”.

#### Raises

**NotImplementedError** – Unable to solve problem using CVXPY

## Returns

A tuple with three items:

1. `x.value` (numpy.ndarray): Optimal value of `x`. `np.nan * np.ones((self.dim,))` if the problem is not solved.
2. `problem.value` (float): Optimal value of the convex program. `np.inf` if the problem is infeasible, `-np.inf` if problem is unbounded, and finite otherwise.
3. `problem_status` (str): Status of the problem

## Return type

tuple

## Notes

This function uses `containment_constraints()` to obtain the list of CVXPY expressions that form the containment constraints on `x`.

### Warning

Please pay attention to the `NotImplementedError` generated by this function. It may be possible to get CVXPY to solve the same problem by switching the solver. For example, consider the following code block.

```
from pycvxset import Polytope
P = Polytope(A=[[1, 1], [-1, -1]], b=[1, 1])
P.cvxpy_args_lp = {'solver': 'CLARABEL'} # Default solver used in
→pycvxset
try:
    print('Is polytope bounded?', P.is_bounded)
except NotImplementedError as err:
    print(str(err))
P.cvxpy_args_lp = {'solver': 'OSQP'}
print('Is polytope bounded?', P.is_bounded)
```

This code block produces the following output:

```
Unable to solve the task (support function evaluation of the set at eta =
→[-0. -1.]). CVXPY returned error:
Solver 'CLARABEL' failed. Try another solver, or solve with verbose=True.
→for more information.
```

```
Is polytope bounded? False
```

`minimize_H_rep()` → None

Remove any redundant inequalities from the halfspace representation of the polytope using `cdd`.

## Raises

**ValueError** – When minimal H-Rep computation fails OR polytope is not bounded!

`minimize_V_rep(prefer_qhull_over_cdd: bool = False, prune_V: bool = False)` → None

Remove any redundant vertices from the vertex representation of the polytope.

## Parameters

- **prefer\_qhull\_over\_cdd** (bool, optional) – When True, `minimize_V_rep` uses `qhull` when possible. Otherwise, we use `cdd`. Default is True.
- **prune\_V** (bool, optional) – When True, prune vertices before minimizing. Defaults to False.

#### Raises

**ValueError** – When minimal V-Rep computation fails!

#### Notes

Use cdd or qhull for the reduction of vertices.

**minimum\_volume\_circumscribing\_ellipsoid()** → tuple[np.ndarray, np.ndarray, np.ndarray | None]

Compute the minimum volume ellipsoid that covers the given polytope (also known as Lowner-John Ellipsoid).

#### Raises

- **ValueError** – When polytope is empty
- **NotImplementedError** – Unable to solve convex problem using CVXPY

#### Returns

A tuple with three items:

1. center (numpy.ndarray): Minimum volume circumscribed ellipsoid's center
2. shape\_matrix (numpy.ndarray): Minimum volume circumscribed ellipsoid's shape matrix
3. sqrt\_shape\_matrix (numpy.ndarray): Minimum volume circumscribed ellipsoid's square root of shape matrix. Returns None if the polytope is not full-dimensional.

#### Return type

tuple

#### Notes

This function requires V-Rep, and will perform a vertex enumeration when a H-Rep polytope is provided.

We solve the SDP for a positive definite matrix  $A \in \mathbb{S}_{++}^n$  and  $b \in \mathbb{R}^n$ ,

$$\begin{aligned} & \text{maximize} && \log \det A^{-1} \\ & \text{subject to} && \|Av_i + b\|_2 \leq 1, \forall \text{ vertices } v_i \text{ of } \mathcal{P} \end{aligned}$$

where the circumscribing ellipsoid is given by  $\{x \mid \|Ax + b\|_2 \leq 1\}$ , and we use the observation that  $\log \det A^{-1} = -\log \det A$ . The center of the ellipsoid is given by  $c = -A^{-1}b$ , and the shape matrix is given by  $Q = (A^T A)^{-1}$ . See [EllipsoidalTbx-Min\_verticesEll] and Section 8.4.1 in [BV04] for more details.

When the polytope is full-dimensional, we can instead solve a second-order cone program (SOCP). Consider the full-dimensional ellipsoid  $\{Lu + c \mid \|u\|_2 \leq 1\}$ , where  $G$  is a square, lower-triangular matrix with positive diagonal entries. Then, we solve the following (equivalent) optimization problem:

$$\begin{aligned} & \text{minimize} && \text{geomean}(L) \\ & \text{subject to} && \text{diag}(L) > 0 \\ & && v_i = Lu_i + c, \forall \text{ vertices } v_i \text{ of } \mathcal{P}, \\ & && u_i \in \mathbb{R}^n, \|u_i\|_2 \leq 1, \end{aligned}$$

with decision variables  $G$ ,  $c$ , and  $u_i$ , and geomean is the geometric mean of the diagonal elements of  $G$ . Here, we use the observation that  $\text{geomean}(L)$  is a monotone function of  $\log \det(GG^T)$  (the volume of the ellipsoid). For the sake of convexity, we solve the following equivalent optimization problem after a change of variables  $L_{\text{inv}} = L^{-1}$  and  $c_{L_{\text{inv}}} = L^{-1}c$ , and substituting for the variables  $u_i$ :

$$\begin{aligned} & \text{maximize} && \text{geomean}(L_{\text{inv}}) \\ & \text{subject to} && \text{diag}(L_{\text{inv}}) > 0 \\ & && \|L_{\text{inv}}v_i - c_{L_{\text{inv}}}\|_2 \leq 1, \forall \text{ vertices } v_i \end{aligned}$$

The center of the ellipsoid is  $c = Lc_{L_{\text{inv}}}$ , and the shape matrix is  $Q = GG^T$ , where  $L = L_{\text{inv}}^{-1}$ .

**minimum\_volume\_circumscribing\_rectangle()** → tuple[np.ndarray, np.ndarray]

Compute the minimum volume circumscribing rectangle for a given polytope

**Raises**

- **ValueError** – When polytope is empty
- **ValueError** – When polytope is unbounded

**Returns**

**A tuple of two items**

1. lb (numpy.ndarray): Lower bound  $l$  on the polytope,  $\mathcal{P} \subseteq \{l\} \oplus \mathbb{R}_{\geq 0}$ .
2. ub (numpy.ndarray): Upper bound  $u$  on the polytope,  $\mathcal{P} \subseteq \{u\} \oplus (-\mathbb{R}_{\geq 0})$ .

**Return type**

tuple

**Notes**

This function accommodates H-Rep and V-Rep. The lower/upper bound for V-Rep is given by an element-wise minimization/maximization operation, while the lower/upper bound for H-Rep is given by an element-wise support computation ( $2 * \text{self.dim}$  linear programs). For a H-Rep polytope, the function uses [support\(\)](#) and is a wrapper for `pycvxset.common.minimum_volume_circumscribing_rectangle()`.

This function is called to check for boundedness of a polytope.

**minus**(*Q: Sequence[float] | Sequence[Sequence[float]] | np.ndarray | ConstrainedZonotope | Ellipsoid | Polytope*) → *Polytope*

Implement - operation (Pontryagin difference when Q is a polytope, translation by -Q when Q is a point)

**Parameters**

**Q** (*Sequence[float] | Sequence[Sequence[float]] | np.ndarray | ConstrainedZonotope | Ellipsoid | Polytope*) – Polytope to be subtracted in Pontryagin difference sense from self or a vector for negative translation of the polytope

**Raises**

- **TypeError** – When Q is neither a Polytope or a point
- **ValueError** – When Q is not of the same dimension as self

**Returns**

The polytope  $\mathcal{R}$  that is the Pontryagin difference of  $P$  and  $Q$  or a negative translation of  $P$  by  $Q$ .

**Return type**

*Polytope*

**Notes**

- *Subtraction with a point:* This function accommodates  $\mathcal{P}$  in H-Rep and V-Rep. See [plus\(\)](#) for more details.
- *Subtraction with a polytope:* This function requires  $\mathcal{P}$  and  $Q$  to be of the same dimension and  $\mathcal{P}$  in H-Rep, and performs halfspace enumerations when  $\mathcal{P}$  is in V-Rep. The H-rep of the polytope  $\mathcal{R}$  is,  $H_{\mathcal{R}} = [\mathcal{P}.A, \mathcal{P}.b - \rho_Q(\mathcal{P}.A)]$  where  $\rho_Q$  is the support function (see [support\(\)](#)). [KG98]

**normalize()** → None

Normalize a H-Rep such that each row of A has unit  $\ell_2$ -norm.

## Notes

This function requires  $P$  to be in H-Rep. If  $P$  is in V-Rep, a halfspace enumeration is performed.

**plot**(*ax*: Axes | Axes3D | None | None = None, *patch\_args*: dict[str, Any] | None = None, *vertex\_args*: dict[str, Any] | None = None, *autoscale\_enable*: bool = True, *decimal\_precision*: int = 3, *enable\_warning*: bool = True) → tuple[Any, Any, Any]

Plot a 2D or 3D polytope.

### Parameters

- **ax** (Axes | Axes3D | None, optional) – Axis on which the patch is to be plotted
- **patch\_args** (dict, optional) – Arguments to pass for plotting faces and edges. See [Matplotlib-patch] for options for patch\_args for 2D and [Matplotlib-Line3DCollection] for options for patch\_args for 3D. Defaults to None, in which case we set edgecolor to black, and facecolor to skyblue.
- **vertex\_args** (dict, optional) – Arguments to pass for plotting vertices. See [Matplotlib-scatter] for options for vertex\_args for 2D and [Matplotlib-Axes3D.scatter] for options for vertex\_args for 3D. Defaults to None, in which case we skip plotting the vertices.
- **autoscale\_enable** (bool, optional) – When set to True, matplotlib adjusts axes to view full polytope. Defaults to True.
- **decimal\_precision** (int, optional) – When plotting a 3D polytope that is in V-Rep and not in H-Rep, we round vertex to the specified precision to avoid numerical issues. Defaults to PLOTTING\_DECIMAL\_PRECISION\_CDD specified in pycvxset.common.constants.
- **enable\_warning** (bool, optional) – Enables the UserWarning. May be turned off if expected. Defaults to True.

### Raises

- **ValueError** – When an incorrect axes is provided.
- **NotImplementedError** – When a polytope.dim >= 4 or == 1 | autoscale\_enable is set to False for 3D plotting.
- **UserWarning** – When an empty polytope or an unbounded polytope is provided.
- **UserWarning** – In 3D plotting, when all faces have less than 3 vertices

### Returns

Tuple with axes containing the polygon, handle for plotting patch, handle for plotting vertices

### Return type

(axes, handle, handle)

## Notes

- plot is a wrapper for `plot2d()` and `plot3d()` functions. See their documentation for more details.
- *Vertex-halfspace enumeration for 2D polytopes*: If a 2D polytope is in H-Rep, vertex enumeration is performed to plot the polytope in 2D. No vertex enumeration is performed for a 2D polytope in V-Rep. This function calls `minimize_V_rep` to simplify plotting.
- *Vertex-halfspace enumeration for 3D polytopes*: The 3D polytope needs to have both V-Rep and H-Rep. Consequently, at least a halfspace/vertex enumeration is performed for a 3D polytope in single representation.
- *Rounding vertices*: This function calls `pycvxset.common.prune_and_round_vertices()` when a 3D polytope in V-Rep is given to be plotted.

- *Handle returned for the patches:* For 2D plot, axis handle of the single patch is returned. For 3D plotting, multiple patches are present, and plot returns the first patch's axis handle. In this case, the order of patches are determined by the use of `pycddlib.copy_input_incidence` from the from the specified (A, b). Also, labeling the first patch is done using `Poly3DCollection`.
- *Disabling face colors:* We can plot the polytope frames without filling it by setting `patch_args['facecolor'] = None` or `patch_args['fill'] = False`. If visual comparison of 3D polytopes is desired, it may be better to plot just the polytope frames instead by setting `patch_args['facecolor'] = None`.

### Warning

This function may produce erroneous-looking plots when visually comparing multiple 3D sets. For more info on matplotlib limitations, see <https://matplotlib.org/stable/api/toolkits/mplot3d/faq.html#my-3d-plot-doesn-t-look-right-at-certain-viewing-angles>. If visual comparison is desired, it may be better to plot just the polytope frame instead by setting `patch_args['facecolor'] = None`.

**plot2d**(*ax*: Axes | Axes3D | None | None = None, *patch\_args*: dict[str, Any] | None = None, *vertex\_args*: dict[str, Any] | None = None, *autoscale\_enable*: bool = True, *enable\_warning*: bool = True) → tuple[Any, Any, Any]

Plot a 2D polytope using matplotlib's `add_patch(Polygon())`.

#### Parameters

- **ax** (Axes | Axes3D | None, optional) – Axis on which the patch is to be plotted
- **patch\_args** (dict, optional) – Arguments to pass for plotting faces and edges. See [Matplotlib-patch] for options for `patch_args`. Defaults to None, in which case we set `edgecolor` to black, and `facecolor` to skyblue.
- **vertex\_args** (dict, optional) – Arguments to pass for plotting vertices. See [Matplotlib-scatter] for options for `vertex_args`. Defaults to None, in which case we skip plotting the vertices.
- **autoscale\_enable** (bool, optional) – When set to True (default), matplotlib adjusts axes to view full polytope.
- **enable\_warning** (bool, optional) – Enables the UserWarning. May be turned off if expected. Defaults to True.

#### Raises

**ValueError** – When a 2D polytope is provided

#### Returns

Tuple with axes containing the polygon, handle for plotting patch, handle for plotting vertices

#### Return type

(axes, handle, handle)

### Notes

This function requires polytope in V-Rep, and performs a vertex enumeration if the polytope is H-Rep.

We sort the vertices in counter-clockwise direction with respect to the centroid, and then plot it using matplotlib's `Polygon`. We can plot just the polytope frames without filling it by setting `patch_args['facecolor'] = None` or `patch_args['fill'] = False`.

**plot3d**(*ax*: Axes | Axes3D | None | None = None, *patch\_args*: dict[str, Any] | None = None, *vertex\_args*: dict[str, Any] | None = None, *autoscale\_enable*: bool = True, *decimal\_precision*: int = 3, *enable\_warning*: bool = True, *prune\_V*: bool = True) → tuple[Any, Any, Any]

Plot a 3D polytope using matplotlib's `Line3DCollection`.



### Parameters

- **ax** (*Axes | Axes3D | None, optional*) – Axes to plot. Defaults to None, in which case a new axes is created. The function assumes that the provided axes was defined with `projection='3d'`.
- **patch\_args** (*dict, optional*) – Arguments to pass for plotting faces and edges. See [\[Matplotlib-Line3DCollection\]](#) for options for `patch_args`. Defaults to None, in which case we set `edgecolor` to black, and `facecolor` to skyblue.
- **vertex\_args** (*dict, optional*) – Arguments to pass for plotting vertices. See [\[Matplotlib-Axes3D.scatter\]](#) for options for `vertex_args`. Defaults to None, in which case we skip plotting the vertices.
- **autoscale\_enable** (*bool, optional*) – When set to True (default), matplotlib adjusts axes to view full polytope.
- **decimal\_precision** (*int, optional*) – When plotting a 3D polytope that is in V-Rep and not in H-Rep, we round vertex to the specified precision to avoid numerical issues. Defaults to `PLOTTING_DECIMAL_PRECISION_CDD` specified in `pycvxset.common.constants`.
- **enable\_warning** (*bool, optional*) – Enables the UserWarning. May be turned off if expected. Defaults to True.
- **prune\_V** (*bool, optional*) – When True, prune vertices before plotting. Defaults to True.

### Raises

- **ValueError** – When either a non-3D polytope is provided.
- **UserWarning** – When all faces have less than 3 vertices
- **NotImplementedError** – `autoscale_enable` needs to be currently enabled for 3D plotting

### Returns

Tuple with axes containing the polygon, handle for plotting patch, handle for plotting vertices

### Return type

(axes, handle, handle)

### Notes

- This function requires polytope in H-Rep and V-Rep, and uses CDD to compute incidences, and vertices. Consequently, at least one halfspace/vertex enumeration is performed.
- Since 3D plotting involves multiple patches, the first patch's axis handle is returned. In this case, the order of patches are determined by the use of `pycddlib.copy_input_incidence` from the specified (A, b).
- This function calls `pycvxset.common.prune_and_round_vertices()` when a 3D polytope in V-Rep is given to be plotted.
- When label is passed in `patch_args`, the label is only applied to the first patch. Subsequent patches will not have a label. Moreover, the first patch is plotted using `Poly3DCollection`, while the subsequent patches are plotted using `Line3DCollection` in this case. Otherwise, when no label is provided, all patches are plotted using `Line3DCollection`.
- We iterate over each halfspace, rotate the halfspace about its centroid so that halfspace is now parallel to XY plane, and then sort the vertices based on their XY coordinates in counter-clockwise direction with respect to the centroid, and then plot it using matplotlib's `Line3DCollection`.

### Warning

This function may produce erroneous-looking plots when visually comparing multiple 3D sets. For more info on matplotlib limitations, see <https://matplotlib.org/stable/api/toolkits/mplot3d/faq.html#my-3d-plot-doesn-t-look-right-at-certain-viewing-angles>. If visual comparison is desired, it may be better to plot just the polytope frame instead by setting `patch_args['facecolor'] = None`.

**plus**(*Q*: *Sequence*[*float*] | *Sequence*[*Sequence*[*float*]] | *np.ndarray* | *Polytope*) → *Polytope*

Add a point or a set to the polytope

#### Parameters

**Q** (*Sequence*[*float*] | *Sequence*[*Sequence*[*float*]] | *np.ndarray* | *Polytope*) – Point or set to add to the polytope.

#### Raises

**ValueError** – When the point dimension does not match the polytope dimension.

#### Returns

*Polytope* which is the sum of self and *Q*.

#### Return type

*Polytope*

### Notes

- Given a polytope  $\mathcal{P}$ , and a set  $Q$ , this function computes the Minkowski sum of  $Q$  and the polytope, defined as  $\mathcal{R} = \{x + q | x \in \mathcal{P}, q \in Q\}$ . On the other hand, when  $Q$  is a point, this function computes the polytope  $\mathcal{R} = \{x + Q | x \in \mathcal{P}\}$ .
- *Addition with a point*: This function allows for  $\mathcal{P}$  to be in V-Rep or in H-Rep. For  $\mathcal{P}$  in V-rep, the polytope  $\mathcal{R}$  is the convex hull of the vertices of  $\mathcal{P}$  shifted by point. Given  $\{v_i\}$  as the collection of vertices of  $\mathcal{P}$ ,  $\mathcal{R} = \text{convexHull}(v_i + \text{point})$ . For  $\mathcal{P}$  in H-rep, the polytope  $\mathcal{R}$  is defined by all points  $r = \text{point} + x$  with  $Ax \leq b$ ,  $A_e x = b_e$ . Thus,  $\mathcal{R} = \{x : Ax \leq b + A\text{point}, A_e x = b_e + A_e \text{point}\}$ .
- *Addition with a polytope*: This function requires self and  $Q$  to be in V-Rep, and performs a vertex enumeration when self or  $Q$  are in H-Rep.. In vertex representation,  $\mathcal{R}$  is the convex hull of the pairwise sum of all combinations of points in  $\mathcal{P}$  and  $Q$ .

**project**(*x*: *Sequence*[*float*] | *Sequence*[*Sequence*[*float*]] | *ndarray*, *p*: *int* | *str* = 2) → *tuple*[*ndarray*, *ndarray*]

Project a point or a collection of points on to a set.

Given a set  $\mathcal{P}$  and a test point  $y \in \mathbb{R}^{\mathcal{P}.\text{dim}}$ , this function solves a convex program,

$$\begin{aligned} &\text{minimize} && \|x - y\|_p \\ &\text{subject to} && x \in \mathcal{P} \end{aligned}$$

#### Parameters

- **points** (*Sequence*[*Sequence*[*float*]] | *np.ndarray*) – Points to project (N times self.dim) with each row as a point.
- **p** (*str* | *int*) – Norm-type. It can be 1, 2, or 'inf'. Defaults to 2, which is the Euclidean norm.

#### Raises

- **ValueError** – Set is empty
- **ValueError** – Dimension mismatch — no. of columns in points is different from self.dim.

- **ValueError** – Points is not convertible into a 2D array
- **NotImplementedError** – Unable to solve problem using CVXPY

#### Returns

A tuple with two items:

1. `projected_point` (numpy.ndarray): Projection point(s) as a 2D numpy.ndarray. Matrix (N times self.dim), where each row is a projection of the point in points to the set  $\mathcal{P}$ .
2. `distance` (numpy.ndarray): Distance(s) as a 1D numpy.ndarray. Vector (N,), where each row is a projection of the point in points to the set  $\mathcal{P}$ .

#### Return type

tuple

#### Notes

- This function allows for  $\mathcal{P}$  to be in V-Rep or in H-Rep.
- Given a polytope  $\mathcal{P}$  in V-Rep and a test point  $y \in \mathbb{R}^{\mathcal{P}.dim}$ , this function solves a convex program with decision variables  $x \in \mathbb{R}^{\mathcal{P}.dim}$  and  $\theta \in \mathbb{R}^{\mathcal{P}.n\_vertices}$ ,

$$\begin{aligned} &\text{minimize} && \|x - y\|_p \\ &\text{subject to} && x = \sum_i \theta_i v_i \\ &&& \sum_i \theta_i = 1, \theta_i \geq 0 \end{aligned}$$

- Given a polytope  $\mathcal{P}$  in H-Rep and a test point  $y \in \mathbb{R}^{\mathcal{P}.dim}$ , this function solves a convex program with a decision variable  $x \in \mathbb{R}^{\mathcal{P}.dim}$ ,

$$\begin{aligned} &\text{minimize} && \|x - y\|_p \\ &\text{subject to} && Ax \leq b \\ &&& A_e x = b_e \end{aligned}$$

**projection**(*project\_away\_dims*: int | Sequence[int] | ndarray) → *Polytope*

Orthogonal projection of a set  $\mathcal{P}$  after removing some user-specified dimensions.

$$\mathcal{R} = \{r \in \mathbb{R}^m \mid \exists v \in \mathbb{R}^{n-m}, \text{Lift}(r, v) \in \mathcal{P}\}$$

Here,  $m = \mathcal{P}.dim - \text{length}(\text{project\_away\_dim})$ , and  $\text{Lift}(r, v)$  lifts (“undo”s the projection) using the appropriate components of  $v$ . This function uses [affine\\_map\(\)](#) to implement the projection by designing an appropriate affine map  $M \in \{0, 1\}^{m \times \mathcal{P}.dim}$  with each row of  $M$  corresponding to some standard axis vector  $e_i \in \mathbb{R}^m$ .

#### Parameters

**project\_away\_dims** (*Sequence[int]* | *np.ndarray*) – Dimensions to projected away in integer interval [0, 1, ..., n - 1].

#### Raises

**ValueError** – When `project_away_dims` are not in the integer interval | All dimensions are projected away

#### Returns

Set obtained via projection.

#### Return type

object

### Returns

m-dimensional set obtained via projection.

### Return type

*Polytope*

### Notes

This function requires  $P$  to be in V-Rep, and performs a vertex enumeration when  $P$  is in H-Rep.

**slice**(*dims*: *int* | *Sequence*[*int*] | *ndarray*, *constants*: *float* | *Sequence*[*float*] | *ndarray*) → *Polytope*

Slice a set restricting certain dimensions to constants.

This function uses `intersection_with_affine_set()` to implement the slicing by designing an appropriate affine set from *dims* and *constants*.

### Parameters

- **dims** (*Sequence*[*int*] | *np.ndarray*) – List of dims to restrict to a constant in the integer interval  $[0, 1, \dots, n - 1]$ .
- **constants** (*float* | *Sequence*[*float*] | *np.ndarray*) – List of constants

### Raises

- **ValueError** – *dims* has entries beyond *n*
- **ValueError** – *dims* and *constants* are not 1D arrays of same size

### Returns

Sliced set.

### Return type

object

### Returns

Polytope that has been sliced at the specified dimensions.

### Return type

*Polytope*

### Notes

This function requires  $\mathcal{P}$  to be in H-Rep, and performs a vertex halfspace when  $\mathcal{P}$  is in V-Rep.

**slice\_then\_projection**(*dims*: *int* | *Sequence*[*int*] | *ndarray*, *constants*: *float* | *Sequence*[*float*] | *ndarray*) → *Polytope*

Wrapper for `slice()` and `projection()`.

The function first restricts a set at certain dimensions to constants, and then projects away those dimensions. Useful for visual inspection of higher dimensional sets.

### Parameters

- **dims** (*Sequence*[*int*] | *np.ndarray*) – List of dims to restrict to a constant in the integer interval  $[0, 1, \dots, \text{dim} - 1]$ , and then project away.
- **constants** (*float* | *Sequence*[*float*] | *np.ndarray*) – List of constants

### Raises

- **ValueError** – *dims* has entries beyond *n*
- **ValueError** – *dims* and *constants* are not 1D arrays of same size
- **ValueError** – When *dims* are not in the integer interval | All dimensions are projected away

### Returns

Sliced then projected set.

**Return type**

object

**Returns**

m-dimensional set obtained via projection after slicing.

**Return type**

*Polytope*

**Notes**

This function requires  $\mathcal{P}$  to be in H-Rep, and performs a vertex halfspace when  $\mathcal{P}$  is in V-Rep.

**support**(*eta*: Sequence[float] | Sequence[Sequence[float]] | ndarray) → tuple[ndarray, ndarray]

Evaluates the support function and support vector of a set.

The support function of a set  $\mathcal{P}$  is defined as  $\rho_{\mathcal{P}}(\eta) = \max_{x \in \mathcal{P}} \eta^{\top} x$ . The support vector of a set  $\mathcal{P}$  is defined as  $\nu_{\mathcal{P}}(\eta) = \arg \max_{x \in \mathcal{P}} \eta^{\top} x$ .

**Parameters**

**eta** (Sequence[float] | Sequence[Sequence[float]] | np.ndarray) – Support directions. Matrix (N times self.dim), where each row is a support direction.

**Raises**

- **ValueError** – Set is empty
- **ValueError** – Mismatch in eta dimension
- **ValueError** – eta is not convertible into a 2D array
- **NotImplementedError** – Unable to solve problem using CVXPY

**Returns**

**A tuple with two items:**

1. support\_function\_evaluations (numpy.ndarray): Support function evaluation(s) as a 2D numpy.ndarray. Vector (N,) with as many rows as eta.
2. support\_vectors (numpy.ndarray): Support vectors as a 2D numpy.ndarray. Matrix N x self.dim with as many rows as eta.

**Return type**

tuple

**Notes**

- This function allows for  $\mathcal{P}$  to be in V-Rep or in H-Rep.
- Given a polytope  $\mathcal{P}$  in V-Rep and a support direction  $\eta \in \mathbb{R}^{\mathcal{P}.dim}$ , this function solves a convex program with decision variables  $x \in \mathbb{R}^{\mathcal{P}.dim}$  and  $\theta \in \mathbb{R}^{\mathcal{P}.n\_vertices}$ ,

$$\begin{aligned} &\text{maximize} && \eta^{\top} x \\ &\text{subject to} && x = \sum_i \theta_i v_i \\ &&& \sum_i \theta_i = 1, \theta_i \geq 0 \end{aligned}$$

- Given a polytope  $\mathcal{P}$  in H-Rep and a support direction  $\eta \in \mathbb{R}^{\mathcal{P}.dim}$ , this function solves a convex program with a decision variable  $x \in \mathbb{R}^{\mathcal{P}.dim}$ ,

$$\begin{array}{ll} \text{maximize} & \eta^\top x \\ \text{subject to} & Ax \leq b \\ & A_e x = b_e \end{array}$$

**volume()** → float

Compute the volume of the polytope using QHull

**Returns**

Volume of the polytope

**Return type**

float

**Notes**

- Works with V-representation: Yes
- Works with H-representation: No
- Performs a vertex-halfspace enumeration when H-rep is provided
- Returns 0 when the polytope is empty
- Requires polytope to be full-dimensional

**property A: ndarray**

Inequality coefficient vectors  $A$  for the polytope  $\{Ax \leq b, A_e x = b_e\}$ .

**Returns**

Inequality coefficient vector (H-Rep).  $A$  is `np.empty((0, self.dim))` for empty polytope.

**Return type**

numpy.ndarray

**Notes**

This function requires the polytope to be in H-Rep, and performs a halfspace enumeration if required.

**property Ae: ndarray**

Equality coefficient vectors  $A_e$  for the polytope  $\{Ax \leq b, A_e x = b_e\}$ .

**Returns**

**Equality coefficient vector (H-Rep).**  $A_e$  is `np.empty((0, self.dim))` for empty or full-dimensional polytope.

**Return type**

numpy.ndarray

**Notes**

This function requires the polytope to be in H-Rep, and performs a halfspace enumeration if required.

**property H: ndarray**

Inequality constraints in halfspace representation  $H=[A, b]$  for the polytope  $\{Ax \leq b, A_e x = b_e\}$ .

**Returns**

H-Rep in  $[A, b]$ .  $H$  is `np.empty((0, self.dim + 1))` for empty polytope.

**Return type**

numpy.ndarray

## Notes

This function requires the polytope to be in H-Rep, and performs a halfspace enumeration if required.

**property He: ndarray**

Equality constraints in halfspace representation  $He=[Ae, be]$  for the polytope  $\{Ax \leq b, A_ex = b_e\}$ .

### Returns

H-Rep in  $[Ae, be]$ . He is `np.empty((0, self.dim + 1))` for empty or full-dimensional polytope.

### Return type

`numpy.ndarray`

## Notes

This function requires the polytope to be in H-Rep, and performs a halfspace enumeration if required.

**property V: ndarray**

Vertex representation ( $V$ ) where the polytope is given by `ConvexHull( $v_i$ )` with  $v_i$  as the rows of  $V = [v_1; v_2; \dots; v_{n_{vertices}}]$ .

### Returns

Vertices of the polytope, arranged row-wise. V is `np.empty((0, self.dim))` if polytope is empty.

### Return type

`numpy.ndarray`

## Notes

This function requires the polytope to be in V-Rep, and performs a vertex enumeration if required.

**property b: ndarray**

Inequality constants  $b$  for the polytope  $\{Ax \leq b, A_ex = b_e\}$ .

### Returns

Inequality constants (H-Rep). b is `np.empty((0,))` for empty polytope.

### Return type

`numpy.ndarray`

## Notes

This function requires the polytope to be in H-Rep, and performs a halfspace enumeration if required.

**property be: ndarray**

Equality constants  $be$  for the polytope  $\{Ax \leq b, A_ex = b_e\}$ .

### Returns

Equality constants (H-Rep). be is `np.empty((0,))` for empty or full-dimensional polytope.

### Return type

`numpy.ndarray`

## Notes

This function requires the polytope to be in H-Rep, and performs a halfspace enumeration if required.

**property cvxpy\_args\_lp: dict[str, Any]**

CVXPY arguments in use when solving a linear program

### Returns

CVXPY arguments in use when solving a linear program. Defaults to dictionary in `py-cvxset.common.DEFAULT_CVXPY_ARGS_LP`.

**Return type**

dict

**property cvxpy\_args\_sdp:** dict[str, Any]

CVXPY arguments in use when solving a semi-definite program

**Returns**

CVXPY arguments in use when solving a semi-definite program. Defaults to dictionary in *pycvxset.common.DEFAULT\_CVXPY\_ARGS\_SDP*.

**Return type**

dict

**property cvxpy\_args\_socp:** dict[str, Any]

CVXPY arguments in use when solving a second-order cone program

**Returns**

CVXPY arguments in use when solving a second-order cone program. Defaults to dictionary in *pycvxset.common.DEFAULT\_CVXPY\_ARGS\_SOCP*.

**Return type**

dict

**property dim:** int

Dimension of the polytope. In H-Rep polytope (A, b), this is the number of columns of A, while in V-Rep, this is the number of components of the vertices.

**Returns**

Dimension of the polytope

**Return type**

int

**property in\_H\_rep:** bool

Check if the polytope have a halfspace representation (H-Rep).

**Returns**

When True, the polytope has halfspace representation (H-Rep). Otherwise, False.

**Return type**

bool

**property in\_V\_rep:** bool

Check if the polytope have a vertex representation (V-Rep).

**Returns**

When True, the polytope has vertex representation (V-Rep). Otherwise, False.

**Return type**

bool

**property is\_bounded:** bool

Check if the polytope is bounded.

**Returns**

True if the polytope is bounded, and False otherwise.

**Return type**

bool

**property is\_empty:** bool

Check if the polytope is empty.

**Returns**

When True, the polytope is empty



**Return type**

bool

**Notes**

This property is well-defined (*self.n\_vertices* == 0) when the polytope is in V-Rep and initialized in the constructor. For H-Rep, it solves a Chebyshev centering problem.

**property is\_full\_dimensional: bool**

Check if the affine dimension of the polytope is the same as the polytope dimension

**Returns**

True when the affine hull containing the polytope has the dimension *self.dim*

**Return type**

bool

**Notes**

This function can have self to be in V-Rep or H-Rep. See Sec. 2.1.3 of [BV04] for discussion on affine dimension.

An empty polytope is full dimensional if dim=0, otherwise it is not full-dimensional.

When the n-dimensional polytope is in V-rep, it is full-dimensional when its affine dimension is n. Recall that, the affine dimension is the dimension of the affine hull of the polytope is the linear subspace spanned by the vectors formed by subtracting the vertices with one of the vertices. Consequently, its dimension is given by the rank of the matrix  $P.V[1:] - P.V[0]$ . When there are fewer than *self.dim* + 1 vertices, we know it is coplanar without checking for matrix rank (simplex needs at least *self.dim* + 1 vertices and that is the polytope with the fewest vertices). For numerical stability, we zero-out all delta vertices below PYCVXSET\_ZERO.

When the n-dimensional polytope is in H-Rep, it is full-dimensional if it can fit a n-dimensional ball of appropriate center and radius inside it (Chebyshev radius).

**property is\_singleton: bool**

Check if the polytope is singleton.

**Returns**

When True, the polytope is singleton.

**Return type**

bool

**Notes**

This property is well-defined (*self.n\_vertices* == 1) when the polytope is in V-Rep and initialized in the constructor. For H-Rep, it solves a minimum\_volume\_circumscribing\_rectangle problem.

**property n\_equalities: int**

Number of linear equality constraints used to define the polytope  $\{Ax \leq b, A_e x = b_e\}$

**Returns**

Number of linear equality constraints

**Return type**

int

**Notes**

A call to this property performs a halfspace enumeration if the polytope is in V-Rep.

**property n\_halfspaces: int**

Number of halfspaces used to define the polytope  $\{Ax \leq b, A_e x = b_e\}$

**Returns**

Number of halfspaces

**Return type**

int

**Notes**

A call to this property performs a halfspace enumeration if the polytope is in V-Rep.

**property n\_vertices: int**

Number of vertices

**Returns**

Number of vertices

**Return type**

int

**Notes**

A call to this property performs a vertex enumeration if the polytope is in H-Rep.

**property type\_of\_set: str**

Return the type of set

**Returns**

Type of the set

**Return type**

str

## PYCVXSET.ELLIPSOID

**class** pycvxset.Ellipsoid

Bases: object

Ellipsoid class.

We can define a bounded, non-empty ellipsoid  $\mathcal{P}$  using **one** of the following combinations:

1.  $(c, Q)$  for a full-dimensional ellipsoid in the **quadratic form**  $\mathcal{P} = \{x \in \mathbb{R}^n \mid (x-c)^T Q^{-1} (x-c) \leq 1\}$  with a n-dimensional positive-definite matrix  $Q$  and a n-dimensional vector  $c$ . Here, pycvxset computes a n-dimensional lower-triangular, square matrix  $G$  that satisfies  $GG^T = Q$ .
2.  $(c, G)$  for a full-dimensional or a degenerate ellipsoid as an **affine transformation of a unit-ball**  $\mathcal{P} = \{x \in \mathbb{R}^n \mid \exists u \in \mathbb{R}^N, x = c + Gu, \|u\|_2 \leq 1\}$  with a n x N matrix  $G$ . Here, pycvxset computes  $Q = GG^T$ .
3.  $(c, r)$  for a ball of radius  $r \geq 0$   $\mathcal{P} = \{x \in \mathbb{R}^n \mid \|x - c\|_2 \leq r\}$ .
4.  $(c)$  for a singleton ellipsoid  $\mathcal{P} = \{c\}$ .

### Parameters

- **c** (*Sequence[float] | numpy.ndarray*) – Center of the ellipsoid  $c$ . Vector of length (self.dim,)
- **Q** (*Sequence[Sequence[float]] | numpy.ndarray, optional*) – Shape matrix of the ellipsoid  $Q$ .  $Q$  must be a positive definite matrix (self.dim times self.dim). (self.dim times self.dim).
- **G** (*Sequence[Sequence[float]] | numpy.ndarray, optional*) – Square root of the shape matrix of the ellipsoid  $G$  that satisfies  $GG^T = Q$ . Need not be a square matrix, but must have self.dim rows. If a singleton must be specified,  $G$  must have zero columns.
- **r** (*scalar, optional*) – Non-negative scalar that provides the radius of the self.dim-dimensional ball.

### Raises

- **ValueError** – When more than one of  $Q, G, r$  was provided
- **ValueError** – When  $c$  or  $Q$  or  $G$  or  $r$  does not satisfy implicit properties

### Notes

1. Empty ellipsoids are not permitted ( $c$  is a required keyword argument).
2. When provided  $G$  is such that  $Q = GG^T$  is positive definite, we overwrite  $G$  with a lower-triangular, square, n-dimensional matrix for consistency. Here,  $G$  has strictly positive diagonal elements, and its determinant is the product of its diagonal elements (see volume computation).
3. We use the eigenvalues of  $Q$  to determine the radii of the maximum volume inscribing ball (Chebyshev radius  $R^-$ ) and the minimum volume circumscribing ball  $R^+ \geq R^-$ .
  - a. The ellipsoid represents a singleton when  $R^+$  is negligible.

b. The ellipsoid is full-dimensional when  $R^-$  is non-trivial.

`__init__(*, c: Sequence[float] | ndarray, Q: Sequence[Sequence[float]] | ndarray) → None`

`__init__(*, c: Sequence[float] | ndarray, G: Sequence[Sequence[float]] | ndarray) → None`

`__init__(*, c: Sequence[float] | ndarray, r: float) → None`

`__init__(*, c: Sequence[float] | ndarray) → None`

Constructor for Ellipsoid

## Methods

<code>__init__()</code>	Constructor for Ellipsoid
<code>affine_hull()</code>	Compute the left null space of self.G to identify the affine hull.
<code>affine_map(M)</code>	Compute the affine transformation of the given ellipsoid based on a given scalar/matrix.
<code>chebyshev_centering()</code>	Compute the Chebyshev center and radius of the ellipsoid.
<code>closest_point(points[, p])</code>	Wrapper for <code>project()</code> to compute the point in the convex set closest to the given point.
<code>containment_constraints(x[, flatten_order])</code>	Get CVXPY constraints for containment of x (a <code>cvxpy.Variable</code> ) in an ellipsoid.
<code>contains(Q)</code>	Check containment of a set or a collection of points in an ellipsoid.
<code>copy()</code>	Create a copy of the ellipsoid.
<code>deflate(cvx_set)</code>	Compute the minimum volume ellipsoid that covers the given set (also known as Lowner-John Ellipsoid).
<code>distance(points[, p])</code>	Wrapper for <code>project()</code> to compute distance of a point to a convex set.
<code>extreme(eta)</code>	Wrapper for <code>support()</code> to compute the extreme point.
<code>inflate(cvx_set)</code>	Compute the maximum volume ellipsoid that fits within the given set.
<code>inflate_ball(cvx_set)</code>	Compute the largest ball (Chebyshev ball) of a given set.
<code>interior_point()</code>	Compute an interior point to the Ellipsoid
<code>intersection_with_affine_set(Ae, be)</code>	Compute the intersection of an ellipsoid with an affine set.
<code>inverse_affine_map_under_invertible_matr</code>	Compute the inverse affine transformation of an ellipsoid based on a given scalar/matrix.
<code>maximum_volume_inscribing_ellipsoid()</code>	Compute the parameters of the maximum volume inscribing ellipsoid for a given ellipsoid.
<code>minimize(x, objective_to_minimize, cvxpy_args)</code>	Solve a convex program with CVXPY objective subject to containment constraints.
<code>minimum_volume_circumscribing_ball()</code>	Compute the parameters of a minimum volume circumscribing ball.
<code>minimum_volume_circumscribing_ellipsoid()</code>	Compute the parameters of the minimum volume circumscribing ellipsoid for a given ellipsoid.
<code>minimum_volume_circumscribing_rectangle()</code>	Compute the minimum volume circumscribing rectangle for a set.
<code>plot([method, ax, direction_vectors, ...])</code>	Plot a polytopic approximation of the set.
<code>plus(point)</code>	Add a point to an ellipsoid
<code>polytopic_inner_approximation(...)</code>	Compute a polytopic inner-approximation of a given set via ray shooting.
<code>polytopic_outer_approximation(...)</code>	Compute a polytopic outer-approximation of a given set via ray shooting.

continues on next page

Table 1 – continued from previous page

<code>project(x[, pl])</code>	Project a point or a collection of points on to a set.
<code>projection(project_away_dims)</code>	Orthogonal projection of a set $\mathcal{P}$ after removing some user-specified dimensions.
<code>quadratic_form_as_a_symmetric_matrix()</code>	Define a $(\text{self.dim} + 1)$ -dimensional symmetric matrix $M$ where $\text{self} = \{x \mid [x, 1] @ M @ [x, 1] \leq 0\}$ .
<code>slice(dims, constants)</code>	Slice a set restricting certain dimensions to constants.
<code>slice_then_projection(dims, constants)</code>	Wrapper for <code>slice()</code> and <code>projection()</code> .
<code>support(eta)</code>	Evaluates the support function and support vector of a set.
<code>volume()</code>	Compute the volume of the ellipsoid.

### Attributes

<code>G</code>	Affine transformation matrix $G$ that satisfies $GG^T = Q$ .
<code>Q</code>	Shape matrix of the ellipsoid $Q$ .
<code>c</code>	Center of the ellipsoid $c$ .
<code>cvxpy_args_lp</code>	CVXPY arguments in use when solving a linear program
<code>cvxpy_args_sdp</code>	CVXPY arguments in use when solving a semi-definite program
<code>cvxpy_args_socp</code>	CVXPY arguments in use when solving a second-order cone program
<code>dim</code>	Dimension of the ellipsoid $dim$ .
<code>is_bounded</code>	Check if the ellipsoid is bounded.
<code>is_empty</code>	Check if the ellipsoid is empty.
<code>is_full_dimensional</code>	Check if the ellipsoid is full-dimensional.
<code>is_singleton</code>	Check if the ellipsoid is a singleton.
<code>latent_dim</code>	Latent dimension of the ellipsoid $dim$ .
<code>type_of_set</code>	Return the type of set

`__init__(*, c: Sequence[float] | ndarray, Q: Sequence[Sequence[float]] | ndarray) → None`

`__init__(*, c: Sequence[float] | ndarray, G: Sequence[Sequence[float]] | ndarray) → None`

`__init__(*, c: Sequence[float] | ndarray, r: float) → None`

`__init__(*, c: Sequence[float] | ndarray) → None`

Constructor for Ellipsoid

`__new__(**kwargs)`

## PYCVXSET.ELLIPSOID (API DETAILS)

*pycvxset.Ellipsoid*

Ellipsoid class.

**class** pycvxset.Ellipsoid.Ellipsoid

Bases: object

Ellipsoid class.

We can define a bounded, non-empty ellipsoid  $\mathcal{P}$  using **one** of the following combinations:

1.  $(c, Q)$  for a full-dimensional ellipsoid in the **quadratic form**  $\mathcal{P} = \{x \in \mathbb{R}^n \mid (x-c)^T Q^{-1}(x-c) \leq 1\}$  with a n-dimensional positive-definite matrix  $Q$  and a n-dimensional vector  $c$ . Here, pycvxset computes a n-dimensional lower-triangular, square matrix  $G$  that satisfies  $GG^T = Q$ .
2.  $(c, G)$  for a full-dimensional or a degenerate ellipsoid as an **affine transformation of a unit-ball**  $\mathcal{P} = \{x \in \mathbb{R}^n \mid \exists u \in \mathbb{R}^N, x = c + Gu, \|u\|_2 \leq 1\}$  with a n x N matrix  $G$ . Here, pycvxset computes  $Q = GG^T$ .
3.  $(c, r)$  for a ball of radius  $r \geq 0$   $\mathcal{P} = \{x \in \mathbb{R}^n \mid \|x - c\|_2 \leq r\}$ .
4.  $(c)$  for a singleton ellipsoid  $\mathcal{P} = \{c\}$ .

### Parameters

- **c** (*Sequence[float] | numpy.ndarray*) – Center of the ellipsoid  $c$ . Vector of length (self.dim,)
- **Q** (*Sequence[Sequence[float]] | numpy.ndarray, optional*) – Shape matrix of the ellipsoid  $Q$ .  $Q$  must be a positive definite matrix (self.dim times self.dim). (self.dim times self.dim).
- **G** (*Sequence[Sequence[float]] | numpy.ndarray, optional*) – Square root of the shape matrix of the ellipsoid  $G$  that satisfies  $GG^T = Q$ . Need not be a square matrix, but must have self.dim rows. If a singleton must be specified,  $G$  must have zero columns.
- **r** (*scalar, optional*) – Non-negative scalar that provides the radius of the self.dim-dimensional ball.

### Raises

- **ValueError** – When more than one of  $Q$ ,  $G$ ,  $r$  was provided
- **ValueError** – When  $c$  or  $Q$  or  $G$  or  $r$  does not satisfy implicit properties

### Notes

1. Empty ellipsoids are not permitted ( $c$  is a required keyword argument).
2. When provided  $G$  is such that  $Q = GG^T$  is positive definite, we overwrite  $G$  with a lower-triangular, square, n-dimensional matrix for consistency. Here,  $G$  has strictly positive diagonal elements, and its determinant is the product of its diagonal elements (see volume computation).

3. We use the eigenvalues of  $Q$  to determine the radii of the maximum volume inscribing ball (Chebyshev radius  $R^-$ ) and the minimum volume circumscribing ball  $R^+ \geq R^-$ .
  - a. The ellipsoid represents a singleton when  $R^+$  is negligible.
  - b. The ellipsoid is full-dimensional when  $R^-$  is non-trivial.

`__init__`(\* , c: *Sequence*[float] | ndarray, Q: *Sequence*[*Sequence*[float]] | ndarray) → None

`__init__`(\* , c: *Sequence*[float] | ndarray, G: *Sequence*[*Sequence*[float]] | ndarray) → None

`__init__`(\* , c: *Sequence*[float] | ndarray, r: float) → None

`__init__`(\* , c: *Sequence*[float] | ndarray) → None

Constructor for Ellipsoid

`affine_hull`() → tuple[ndarray | None, ndarray | None]

Compute the left null space of self.G to identify the affine hull. Affine hull is the entire self.dim-dimensional space when self is full-dimensional.

#### Returns

(Ae, be) defining the affine set  $\{x \mid Ae x = be\}$ , or (None, None) when full-dimensional.

#### Return type

tuple

`affine_map`(M: int | float | *Sequence*[*Sequence*[float]] | np.ndarray) → *Ellipsoid*

Compute the affine transformation of the given ellipsoid based on a given scalar/matrix.

#### Parameters

**M**(int | float | *Sequence*[*Sequence*[float]] | np.ndarray) – Scalar or matrix (m times self.dim) for the affine map.

#### Raises

- **ValueError** – When M is not convertible into a 2D numpy array of float
- **ValueError** – When M has columns not equal to self.dim

#### Returns

Affine transformation of the given ellipsoid  $\mathcal{R} = M\mathcal{P} = \{Mx \mid x \in \mathcal{P}\}$

#### Return type

*Ellipsoid*

`chebyshev_centering`() → tuple[ndarray, float]

Compute the Chebyshev center and radius of the ellipsoid.

#### Returns

(center, radius) of the maximum volume inscribed ball.

#### Return type

tuple

`closest_point`(points: *Sequence*[*Sequence*[float]] | np.ndarray, p: int | str = 2) → np.ndarray

Wrapper for `project()` to compute the point in the convex set closest to the given point.

#### Parameters

- **points** (*Sequence*[*Sequence*[float]] | np.ndarray) – Points to project. Matrix (N times self.dim), where each row is a point.
- **p** (str | int) – Norm-type. It can be 1, 2, or ‘inf’. Defaults to 2.

#### Returns

Projection of points to the set as a 2D numpy.ndarray. These arrays have as many rows as points.

#### Return type

numpy.ndarray

## Notes

For more detailed description, see documentation for `project()` function.

**containment\_constraints**(*x*: `cvxpy.Variable`, *flatten\_order*: `Literal['F', 'C'] = 'F'`) → `tuple[list[cvxpy.Constraint], cvxpy.Variable | None]`

Get CVXPY constraints for containment of *x* (a `cvxpy.Variable`) in an ellipsoid.

### Parameters

- ***x*** (`cvxpy.Variable`) – CVXPY variable to be optimized
- ***flatten\_order*** (`Literal["F", "C"]`) – Order to use for flatten (choose between “F”, “C”). Defaults to “F”, which implements column-major flatten. In 2D, column-major flatten results in stacking rows horizontally to achieve a single horizontal row.

### Returns

A tuple with two items:

1. `constraint_list` (`list[cvxpy.Constraint]`): CVXPY constraints for the containment of *x* in the ellipsoid.
2. `xi` (`cvxpy.Variable` | `None`): CVXPY variable representing the latent dimension variable with length `G.shape[1]`. It is `None` when the ellipsoid is a singleton.

### Return type

`tuple`

**contains**(*Q*: `Sequence[float]` | `Sequence[Sequence[float]]` | `np.ndarray` | `Polytope` | `Ellipsoid`) → `bool` | `np.ndarray`

Check containment of a set or a collection of points in an ellipsoid.

### Parameters

***Q*** (`Sequence[float]` | `Sequence[Sequence[float]]` | `np.ndarray` | `Polytope` | `Ellipsoid`) – Polytope/Ellipsoid or a collection of points (each row is a point) to be tested for containment. When providing a collection of points, *Q* is a matrix (*N* times `self.dim`) with each row is a point.

### Raises

- **ValueError** – Test point(s) are NOT of the same dimension
- **ValueError** – Test point(s) can not be converted into a 2D numpy array of floats
- **ValueError** – *Q* is a constrained zonotope
- **NotImplementedError** – Unable to perform ellipsoidal containment check using CVXPY

### Returns

An element of the array is **True** if the point is in the ellipsoid, with as many elements as the number of rows in `test_points`.

### Return type

`bool` or `numpy.ndarray[bool]`

## Notes

- **Containment of a polytope:** This function requires the polytope *Q* to be in V-Rep. If *Q* is in H-Rep, a vertex enumeration is performed. This function then checks if all vertices of *Q* are in the given ellipsoid, which occurs if and only if the polytope is contained within the ellipsoid [BV04].
- **Containment of an ellipsoid:** This function solves a semi-definite program (S-procedure) [BV04].
- **Containment of points:** For each point *v*, the function checks if there is a  $u \in \mathbb{R}^{\mathcal{P}.dim}$  such that  $\|u\|_2 \leq 1$  and  $Gu + c = v$  [BV04]. This can be efficiently done via `numpy.linalg.lstsq`.



**copy()** → *Ellipsoid*

Create a copy of the ellipsoid. Copy (c, G) to preserve degenerate ellipsoids.

**classmethod deflate**(*cvx\_set*: *Ellipsoid* | *Polytope*) → *Ellipsoid*

Compute the minimum volume ellipsoid that covers the given set (also known as Lowner-John Ellipsoid).

**Parameters**

**cvx\_set** (*Ellipsoid* / *Polytope*) – Set to be circumscribed.

**Returns**

Minimum volume circumscribing ellipsoid

**Return type**

*Ellipsoid*

**Notes**

This function is a wrapper for `minimum_volume_circumscribing_ellipsoid()` of the set `set_to_be_centered`. Please check that function for more details including raising exceptions. [EllipsoidalTbx-Min\_verticesolEll]

**distance**(*points*: *Sequence[Sequence[float]]* | *np.ndarray*, *p*: *int* | *str* = 2) → *np.ndarray*

Wrapper for `project()` to compute distance of a point to a convex set.

**Parameters**

- **points** (*Sequence[Sequence[float]]* / *np.ndarray*) – Points to project. Matrix (N times self.dim), where each row is a point.
- **p** (*int* / *str*) – Norm-type. It can be 1, 2, or 'inf'. Defaults to 2.

**Returns**

Distance of points to the set as a 1D numpy.ndarray. These arrays have as many rows as points.

**Return type**

numpy.ndarray

**Notes**

For more detailed description, see documentation for `project()` function.

**extreme**(*eta*: *Sequence[Sequence[float]]* | *np.ndarray*) → *np.ndarray*

Wrapper for `support()` to compute the extreme point.

**Parameters**

**eta** (*Sequence[Sequence[float]]* / *np.ndarray*) – Support directions. Matrix (N times self.dim), where each row is a support direction.

**Returns**

Support vector evaluation(s) as a 2D numpy.ndarray. The array has as many rows as eta.

**Return type**

numpy.ndarray

**Notes**

For more detailed description, see documentation for `support()` function.

**classmethod inflate**(*cvx\_set*: *ConstrainedZonotope* | *Ellipsoid* | *Polytope*) → *Ellipsoid*

Compute the maximum volume ellipsoid that fits within the given set.

**Parameters**

**cvx\_set** (*ConstrainedZonotope* / *Ellipsoid* / *Polytope*) – Set to be inscribed.

### Returns

Maximum volume inscribing ellipsoid

### Return type

*Ellipsoid*

### Notes

This function is a wrapper for `maximum_volume_inscribing_ellipsoid()` of the set `set_to_expand_within`. Please check that function for more details including raising exceptions. [EllipsoidalTbx-MinVolEll]

**classmethod** `inflate_ball(cvx_set: ConstrainedZonotope | Ellipsoid | Polytope) → Ellipsoid`

Compute the largest ball (Chebyshev ball) of a given set.

### Parameters

**cvx\_set** (`ConstrainedZonotope` | `Ellipsoid` | `Polytope`) – Set to compute Chebyshev ball for.

### Returns

Maximum volume inscribing ellipsoid

### Return type

*Ellipsoid*

### Notes

This function is a wrapper for `chebyshev_center()` of attr: `set_to_be_centered`. Please check that function for more details including raising exceptions.

**interior\_point()** → `ndarray`

Compute an interior point to the Ellipsoid

### Returns

center of the ellipsoid, which is an interior point.

### Return type

`np.ndarray`

**intersection\_with\_affine\_set**(`Ae: Sequence[Sequence[float]] | np.ndarray, be: Sequence[float] | np.ndarray`) → *Ellipsoid*

Compute the intersection of an ellipsoid with an affine set.

### Parameters

- **Ae** (`Sequence[Sequence[float]]` | `np.ndarray`) – Equality coefficient matrix (N times self.dim) that define the affine set  $\{x | A_e x = b_e\}$ .
- **be** (`Sequence[float]` | `np.ndarray`) – Equality constant vector (N,) that define the affine set  $\{x | A_e x = b_e\}$ .

### Raises

**ValueError** – When the number of columns in Ae is different from self.dim

### Returns

The intersection of an ellipsoid with the affine set.

### Return type

*Ellipsoid*

### Notes

This function implements imposes the constraints  $\{A_e x = b_e\}$  as constraints in the latent dimension of the ellipsoid —  $A_e(G\xi + c) = b_e$  for every feasible  $\xi$ .

**inverse\_affine\_map\_under\_invertible\_matrix**(*M*: *int* | *float* | *Sequence*[*Sequence*[*float*]] | *np.ndarray*) → *Ellipsoid*

Compute the inverse affine transformation of an ellipsoid based on a given scalar/matrix.

**Parameters**

**M** (*int* | *float* | *Sequence*[*Sequence*[*float*]] | *np.ndarray*) – Scalar or invertible square matrix for the affine map

**Raises**

- **TypeError** – When *M* is not convertible into a 2D square numpy matrix
- **TypeError** – When *M* is not invertible

**Returns**

Inverse affine transformation of the given ellipsoid  $\mathcal{R} = \mathcal{P}M = \{x | Mx \in \mathcal{P}\}$

**Return type**

*Ellipsoid*

**Notes**

1. Since *M* is invertible,  $\mathcal{R}$  is also a bounded ellipsoid.

**maximum\_volume\_inscribing\_ellipsoid**() → tuple[*ndarray*, *ndarray*, *ndarray*]

Compute the parameters of the maximum volume inscribing ellipsoid for a given ellipsoid.

**Returns**

(center, Q, G) describing the ellipsoid.

**Return type**

tuple

**minimize**(*x*: *cvxpy.Variable*, *objective\_to\_minimize*: *cvxpy.Expression*, *cvxpy\_args*: dict[*str*, *Any*], *task\_str*: *str* = "") → tuple[*np.ndarray*, *float*, *str*]

Solve a convex program with CVXPY objective subject to containment constraints.

**Parameters**

- **x** (*cvxpy.Variable*) – CVXPY variable to be optimized
- **objective\_to\_minimize** (*cvxpy.Expression*) – CVXPY expression to be minimized
- **cvxpy\_args** (*dict*) – CVXPY arguments to be passed to the solver
- **task\_str** (*str*, *optional*) – Task string to be used in error messages. Defaults to “.”

**Raises**

**NotImplementedError** – Unable to solve problem using CVXPY

**Returns**

**A tuple with three items:**

1. *x.value* (*numpy.ndarray*): Optimal value of *x*. *np.nan* \* *np.ones*((*self.dim*,)) if the problem is not solved.
2. *problem.value* (*float*): Optimal value of the convex program. *np.inf* if the problem is infeasible, *-np.inf* if problem is unbounded, and finite otherwise.
3. *problem\_status* (*str*): Status of the problem

**Return type**

tuple

## Notes

This function uses `containment_constraints()` to obtain the list of CVXPY expressions that form the containment constraints on  $x$ .

### Warning

Please pay attention to the `NotImplementedError` generated by this function. It may be possible to get CVXPY to solve the same problem by switching the solver. For example, consider the following code block.

```
from pycvxset import Polytope
P = Polytope(A=[[1, 1], [-1, -1]], b=[1, 1])
P.cvxpy_args_lp = {'solver': 'CLARABEL'}    # Default solver used in
→pycvxset
try:
    print('Is polytope bounded?', P.is_bounded)
except NotImplementedError as err:
    print(str(err))
P.cvxpy_args_lp = {'solver': 'OSQP'}
print('Is polytope bounded?', P.is_bounded)
```

This code block produces the following output:

```
Unable to solve the task (support function evaluation of the set at eta =
→[-0. -1.]). CVXPY returned error:
Solver 'CLARABEL' failed. Try another solver, or solve with verbose=True
→for more information.
```

```
Is polytope bounded? False
```

**minimum\_volume\_circumscribing\_ball()** → tuple[ndarray, float]

Compute the parameters of a minimum volume circumscribing ball.

#### Returns

(center, radius) for the minimum-volume circumscribing ball.

#### Return type

tuple

**minimum\_volume\_circumscribing\_ellipsoid()** → tuple[ndarray, ndarray, ndarray]

Compute the parameters of the minimum volume circumscribing ellipsoid for a given ellipsoid.

#### Returns

(center, Q, G) describing the ellipsoid.

#### Return type

tuple

**minimum\_volume\_circumscribing\_rectangle()** → tuple[np.ndarray, np.ndarray]

Compute the minimum volume circumscribing rectangle for a set.

#### Raises

**ValueError** – Solver error or set is empty!

#### Returns

A tuple of two elements

- lb (numpy.ndarray): Lower bound  $l$  on the set,  $\mathcal{P} \subseteq \{l\} \oplus \mathbb{R}_{\geq 0}$ .
- ub (numpy.ndarray): Upper bound  $u$  on the set,  $\mathcal{P} \subseteq \{u\} \oplus (-\mathbb{R}_{\geq 0})$ .

**Return type**  
tuple

### Notes

This function computes the lower/upper bound by an element-wise support computation ( $2n$  linear programs), where  $n$  is `attr:self.dim`. To reuse the [support\(\)](#) function for the lower bound computation, we solve the optimization for each  $i \in \{1, 2, \dots, n\}$ ,

$$\inf_{x \in \mathcal{P}} e_i^\top x = - \sup_{x \in \mathcal{P}} -e_i^\top x = -\rho_{\mathcal{P}}(-e_i),$$

where  $e_i \in \mathbb{R}^n$  denotes the standard coordinate vector, and  $\rho_{\mathcal{P}}$  is the support function of  $\mathcal{P}$ .

**plot**(*method*: str = 'inner', *ax*: Axes | Axes3D | None | None = None, *direction\_vectors*: Sequence[Sequence[float]] | np.ndarray | None = None, *n\_vertices*: int | None = None, *n\_halfspaces*: int | None = None, *patch\_args*: dict[str, Any] | None = None, *vertex\_args*: dict[str, Any] | None = None, *center\_args*: dict[str, Any] | None = None, *autoscale\_enable*: bool = True, *decimal\_precision*: int = 3, *enable\_warning*: bool = True) → tuple[Any, ...]

Plot a polytopical approximation of the set.

### Parameters

- **method** (str, optional) – Type of polytopical approximation to use. Can be [“inner” or “outer”]. Defaults to “inner”.
- **ax** (Axes | Axes3D | None, optional) – Axis on which the patch is to be plotted
- **direction\_vectors** (Sequence[Sequence[float]] | np.ndarray, optional) – Directions to use when performing ray shooting. Matrix ( $N$  times `self.dim`) for some  $N \geq 1$ . Defaults to None, in which case we use [pycvxset.common.spread\\_points\\_on\\_a\\_unit\\_sphere\(\)](#) to compute the direction vectors.
- **n\_vertices** (int, optional) – Number of vertices to use when computing the polytopical inner-approximation. Ignored if method is “outer” or when `direction_vectors` are provided. More than `n_vertices` may be used in some cases (see notes). Defaults to None.
- **n\_halfspaces** (int, optional) – Number of halfspaces to use when computing the polytopical outer-approximation. Ignored if method is “outer” or when `direction_vectors` are provided. More than `n_halfspaces` may be used in some cases (see notes). Defaults to None.
- **patch\_args** (dict, optional) – Arguments to pass for plotting faces and edges. See [pycvxset.Polytope.Polytope.plot\(\)](#) for more details. Defaults to None.
- **vertex\_args** (dict, optional) – Arguments to pass for plotting vertices. See [pycvxset.Polytope.Polytope.plot\(\)](#) for more details. Defaults to None.
- **center\_args** (dict, optional) – For ellipsoidal set, arguments to pass to scatter plot for the center. If a label is desired, pass it in `center_args`.
- **autoscale\_enable** (bool, optional) – When set to True, matplotlib adjusts axes to view full polytope. See [pycvxset.Polytope.Polytope.plot\(\)](#) for more details. Defaults to True.
- **decimal\_precision** (int, optional) – When plotting a 3D polytope that is in V-Rep and not in H-Rep, we round vertex to the specified precision to avoid numerical issues. Defaults to `PLOTTING_DECIMAL_PRECISION_CDD` specified in `pycvxset.common.constants`.
- **enable\_warning** (bool, optional) – Enables the UserWarning. May be turned off if expected. Defaults to True.

### Returns

See [pycvxset.Polytope.Polytope.plot\(\)](#) for details.

**Return type**  
tuple

### Notes

This function is a wrapper for `polytopic_inner_approximation()` and `polytopic_outer_approximation()` for more details in polytope construction.

**plus**(*point*: *Sequence[float] | np.ndarray*) → *Ellipsoid*

Add a point to an ellipsoid

### Parameters

**point** (*Sequence[float] | np.ndarray*) – Vector (self.dim,) that describes the point to be added.

### Raises

- **ValueError** – point is a set (ConstrainedZonotope or Ellipsoid or Polytope)
- **TypeError** – point can not be converted into a numpy array of float
- **ValueError** – point can not be converted into a 1D numpy array of float
- **ValueError** – Mismatch in dimension

### Returns

Sum of the ellipsoid and the point.

### Return type

*Ellipsoid*

**polytopic\_inner\_approximation**(*direction\_vectors*: *Sequence[Sequence[float]] | np.ndarray | None* = *None*, *n\_vertices*: *int | None* = *None*, *verbose*: *bool* = *False*, *enable\_warning*: *bool* = *True*) → *Polytope*

Compute a polytopic inner-approximation of a given set via ray shooting.

### Parameters

- **cvx\_set** (*ConstrainedZonotope | Ellipsoid | Polytope*) – Set to be approximated,
- **direction\_vectors** (*Sequence[Sequence[float]] | np.ndarray, optional*) – Directions to use when performing ray shooting. Matrix ( $N$  times self.dim) for some  $N \geq 1$ . Defaults to None.
- **n\_vertices** (*int, optional*) – Number of vertices to be used for the inner-approximation. *n\_vertices* is overridden whenever *direction\_vectors* are provided. Defaults to None.
- **verbose** (*bool, optional*) – If true, `pycvxset.common.spread_points_on_a_unit_sphere()` is passed with *verbose*. Defaults to False.
- **enable\_warning** (*bool, optional*) – Enables the UserWarning. May be turned off if expected. Defaults to True.

### Returns

Polytopic inner-approximation in V-Rep of a given set with *n\_vertices* no smaller than user-provided *n\_vertices*.

### Return type

*Polytope*

## Notes

We compute the polytope using `extreme()` evaluated along the direction vectors computed by `pycvxset.common.spread_points_on_a_unit_sphere()`. When `direction_vectors` is `None` and `n_vertices` is `None`, we select `n_vertices = 2self.dim + 2self.dimSPOAUS_DIRECTIONS_PER_QUADRANT` (as defined in `pycvxset.common.constants()`). [BV04] The function also uses `pycvxset.common.make_aspect_ratio_equal()` to account for possibly non-symmetric sets.

**polytopic\_outer\_approximation**(*direction\_vectors*: *Sequence[Sequence[float]] | np.ndarray | None = None*, *n\_halfspaces*: *int | None = None*, *verbose*: *bool = False*, *enable\_warning*: *bool = True*) → *Polytope*

Compute a polytopic outer-approximation of a given set via ray shooting.

### Parameters

- **cvx\_set** (*ConstrainedZonotope | Ellipsoid | Polytope*) – Set to be approximated,
- **direction\_vectors** (*Sequence[Sequence[float]] | np.ndarray, optional*) – Directions to use when performing ray shooting. Matrix ( $N$  times  $\text{self.dim}$ ) for some  $N \geq 1$ . Defaults to `None`.
- **n\_halfspaces** (*int, optional*) – Number of halfspaces to be used for the inner-approximation. `n_vertices` is overridden whenever `direction_vectors` are provided. Defaults to `None`.
- **verbose** (*bool, optional*) – If `true`, `pycvxset.common.spread_points_on_a_unit_sphere()` is passed with `verbose`. Defaults to `False`.
- **enable\_warning** (*bool, optional*) – Enables the `UserWarning`. May be turned off if expected. Defaults to `True`.

### Returns

Polytopic outer-approximation in H-Rep of a given set with `n_halfspaces` no smaller than user-provided `n_vertices`.

### Return type

*Polytope*

## Notes

We compute the polytope using `support()` evaluated along the direction vectors computed by `pycvxset.common.spread_points_on_a_unit_sphere()`. When `direction_vectors` is `None` and `n_halfspaces` is `None`, we select `n_halfspaces = 2self.dim + 2self.dimSPOAUS_DIRECTIONS_PER_QUADRANT` (as defined in `pycvxset.common.constants()`). [BV04]

**project**(*x*: *Sequence[float] | Sequence[Sequence[float]] | ndarray*, *p*: *int | str = 2*) → *tuple[ndarray, ndarray]*

Project a point or a collection of points on to a set.

Given a set  $\mathcal{P}$  and a test point  $y \in \mathbb{R}^{\mathcal{P}.\text{dim}}$ , this function solves a convex program,

$$\begin{aligned} &\text{minimize} && \|x - y\|_p \\ &\text{subject to} && x \in \mathcal{P} \end{aligned}$$

### Parameters

- **points** (*Sequence[Sequence[float]] | np.ndarray*) – Points to project ( $N$  times  $\text{self.dim}$ ) with each row as a point.

- **p** (*str* / *int*) – Norm-type. It can be 1, 2, or ‘inf’. Defaults to 2, which is the Euclidean norm.

#### Raises

- **ValueError** – Set is empty
- **ValueError** – Dimension mismatch — no. of columns in points is different from `self.dim`.
- **ValueError** – Points is not convertible into a 2D array
- **NotImplementedError** – Unable to solve problem using CVXPY

#### Returns

A tuple with two items:

1. `projected_point` (`numpy.ndarray`): Projection point(s) as a 2D `numpy.ndarray`. Matrix ( $N$  times `self.dim`), where each row is a projection of the point in points to the set  $\mathcal{P}$ .
2. `distance` (`numpy.ndarray`): Distance(s) as a 1D `numpy.ndarray`. Vector ( $N$ ), where each row is a projection of the point in points to the set  $\mathcal{P}$ .

#### Return type

tuple

#### Notes

For a point  $y \in \mathbb{R}^{\mathcal{P}.dim}$  and an ellipsoid  $\mathcal{P} = \{Gu + c \mid \|u\|_2 \leq 1\}$  with  $GG^T = Q$ , this function solves a convex program with decision variables  $x, u \in \mathbb{R}^{\mathcal{P}.dim}$ ,

$$\begin{aligned} &\text{minimize} && \|x - y\|_p \\ &\text{subject to} && x = Gu + c \\ &&& \|u\|_2 \leq 1 \end{aligned}$$

**projection**(*project\_away\_dims*: *int* | *Sequence[int]*) → *Ellipsoid*

Orthogonal projection of a set  $\mathcal{P}$  after removing some user-specified dimensions.

$$\mathcal{R} = \{r \in \mathbb{R}^m \mid \exists v \in \mathbb{R}^{n-m}, \text{Lift}(r, v) \in \mathcal{P}\}$$

Here,  $m = \mathcal{P}.dim - \text{length}(\text{project\_away\_dim})$ , and  $\text{Lift}(r, v)$  lifts (“undo”s the projection) using the appropriate components of  $v$ . This function uses [affine\\_map\(\)](#) to implement the projection by designing an appropriate affine map  $M \in \{0, 1\}^{m \times \mathcal{P}.dim}$  with each row of  $M$  corresponding to some standard axis vector  $e_i \in \mathbb{R}^m$ .

#### Parameters

**project\_away\_dims** (*Sequence[int]* | *np.ndarray*) – Dimensions to projected away in integer interval  $[0, 1, \dots, n - 1]$ .

#### Raises

**ValueError** – When `project_away_dims` are not in the integer interval | All dimensions are projected away

#### Returns

Set obtained via projection.

#### Return type

object

#### Returns

$m$ -dimensional set obtained via projection.

#### Return type

*Ellipsoid*



**quadratic\_form\_as\_a\_symmetric\_matrix()** → ndarray

Define a (self.dim + 1)-dimensional symmetric matrix  $M$  where  $\text{self} = \{x \mid [x, 1] @ M @ [x, 1] \leq 0\}$ . Here, when  $Q$  is not positive definite, we use pseudo-inverse of  $Q$ .

**Returns**

(self.dim + 1) x (self.dim + 1) symmetric matrix defining the quadratic form.

**Return type**

numpy.ndarray

**slice**(dims: int | Sequence[int] | ndarray, constants: float | Sequence[float] | ndarray) → *Ellipsoid*

Slice a set restricting certain dimensions to constants.

This function uses [intersection\\_with\\_affine\\_set\(\)](#) to implement the slicing by designing an appropriate affine set from dims and constants.

**Parameters**

- **dims** (Sequence[int] | np.ndarray) – List of dims to restrict to a constant in the integer interval  $[0, 1, \dots, n - 1]$ .
- **constants** (float | Sequence[float] | np.ndarray) – List of constants

**Raises**

- **ValueError** – dims has entries beyond  $n$
- **ValueError** – dims and constants are not 1D arrays of same size

**Returns**

Sliced set.

**Return type**

object

**Returns**

Ellipsoid that has been sliced at the specified dimensions.

**Return type**

*Ellipsoid*

**Notes**

- 

**slice\_then\_projection**(dims: int | Sequence[int], constants: int | Sequence[int]) → *Ellipsoid*

Wrapper for [slice\(\)](#) and [projection\(\)](#).

The function first restricts a set at certain dimensions to constants, and then projects away those dimensions. Useful for visual inspection of higher dimensional sets.

**Parameters**

- **dims** (Sequence[int] | np.ndarray) – List of dims to restrict to a constant in the integer interval  $[0, 1, \dots, \text{dim} - 1]$ , and then project away.
- **constants** (float | Sequence[float] | np.ndarray) – List of constants

**Raises**

- **ValueError** – dims has entries beyond  $n$
- **ValueError** – dims and constants are not 1D arrays of same size
- **ValueError** – When dims are not in the integer interval | All dimensions are projected away

**Returns**

Sliced then projected set.

**Return type**

object

**Returns**

m-dimensional set obtained via projection after slicing.

**Return type**

*Ellipsoid*

**support**(*eta*: ndarray | Sequence[float] | Sequence[Sequence[float]]) → tuple[ndarray, ndarray]

Evaluates the support function and support vector of a set.

The support function of a set  $\mathcal{P}$  is defined as  $\rho_{\mathcal{P}}(\eta) = \max_{x \in \mathcal{P}} \eta^\top x$ . The support vector of a set  $\mathcal{P}$  is defined as  $\nu_{\mathcal{P}}(\eta) = \arg \max_{x \in \mathcal{P}} \eta^\top x$ .

**Parameters**

**eta** (Sequence[float] | Sequence[Sequence[float]] | np.ndarray) – Support directions. Matrix (N times self.dim), where each row is a support direction.

**Raises**

- **ValueError** – Set is empty
- **ValueError** – Mismatch in eta dimension
- **ValueError** – eta is not convertible into a 2D array
- **NotImplementedError** – Unable to solve problem using CVXPY

**Returns**

A tuple with two items:

1. support\_function\_evaluations (numpy.ndarray): Support function evaluation(s) as a 2D numpy.ndarray. Vector (N,) with as many rows as eta.
2. support\_vectors (numpy.ndarray): Support vectors as a 2D numpy.ndarray. Matrix N x self.dim with as many rows as eta.

**Notes**

Using duality, the support function and vector of an ellipsoid has a closed-form expressions. For a support direction  $\eta \in \mathbb{R}^{\mathcal{P}.\text{dim}}$  and an ellipsoid  $\mathcal{P} = \{Gu + c \mid \|u\|_2 \leq 1\}$  with  $GG^T = Q$ ,

$$\begin{aligned}\rho_{\mathcal{P}}(\eta) &= \eta^\top c + \sqrt{\eta^\top Q \eta} = \eta^\top c + \|G^T \eta\|_2 \\ \nu_{\mathcal{P}}(\eta) &= c + \frac{GG^\top \eta}{\|G^T \eta\|_2} = c + \frac{Q\eta}{\|G^T \eta\|_2}\end{aligned}$$

For degenerate (not full-dimensional) ellipsoids and  $\eta$  not in the low-dimensional affine hull containing the ellipsoid,

$$\begin{aligned}\rho_{\mathcal{P}}(\eta) &= \eta^\top c \\ \nu_{\mathcal{P}}(\eta) &= c\end{aligned}$$

**Return type**

tuple

**volume**() → float

Compute the volume of the ellipsoid.

**Returns**

Volume of the ellipsoid.

**Return type**

float

## Notes

Volume of the ellipsoid is zero if it is not full-dimensional. For full-dimensional ellipsoid, we used the following observations:

1. from [BV04], the volume of an ellipsoid is proportional to  $\det(G)$ .
2. Square-root of the determinant of the shape matrix coincides with the determinant of  $G$
3. Since  $G$  is lower-triangular, its determinant is the product of its diagonal elements.

### property **G**: ndarray

Affine transformation matrix  $G$  that satisfies  $GG^T = Q$ .

#### Returns

Generator matrix.

#### Return type

numpy.ndarray

### property **Q**: ndarray

Shape matrix of the ellipsoid  $Q$ .

#### Returns

Shape matrix.

#### Return type

numpy.ndarray

### property **c**: ndarray

Center of the ellipsoid  $c$ .

#### Returns

Center vector.

#### Return type

numpy.ndarray

### property **cvxpy\_args\_lp**: dict[str, Any]

CVXPY arguments in use when solving a linear program

#### Returns

CVXPY arguments in use when solving a linear program. Defaults to dictionary in *pycvxset.common.DEFAULT\_CVXPY\_ARGS\_LP*.

#### Return type

dict

### property **cvxpy\_args\_sdp**: dict[str, Any]

CVXPY arguments in use when solving a semi-definite program

#### Returns

CVXPY arguments in use when solving a semi-definite program. Defaults to dictionary in *pycvxset.common.DEFAULT\_CVXPY\_ARGS\_SDP*.

#### Return type

dict

### property **cvxpy\_args\_socp**: dict[str, Any]

CVXPY arguments in use when solving a second-order cone program

#### Returns

CVXPY arguments in use when solving a second-order cone program. Defaults to dictionary in *pycvxset.common.DEFAULT\_CVXPY\_ARGS\_SOCP*.

#### Return type

dict

**property dim: int**

Dimension of the ellipsoid *dim*.

**Returns**

Dimension of the ellipsoid.

**Return type**

int

**property is\_bounded: bool**

Check if the ellipsoid is bounded. Always True by construction.

**property is\_empty: bool**

Check if the ellipsoid is empty. Always False by construction.

**Returns**

Always False for ellipsoids since we require non-empty ellipsoids.

**Return type**

bool

**property is\_full\_dimensional: bool**

Check if the ellipsoid is full-dimensional.

**Returns**

True if full-dimensional.

**Return type**

bool

**property is\_singleton: bool**

Check if the ellipsoid is a singleton.

**Returns**

True if the ellipsoid is a singleton.

**Return type**

bool

**property latent\_dim: int**

Latent dimension of the ellipsoid *dim*.

**Returns**

Latent dimension of the ellipsoid.

**Return type**

int

**property type\_of\_set: str**

Return the type of set

**Returns**

Type of the set

**Return type**

str

## PYCVXSET.CONSTRAINEDZONOTOPE

**class** pycvxset.ConstrainedZonotope

Bases: object

Constrained zonotope class

Constrained zonotope defines a polytope in the working dimension  $\mathbb{R}^n$  as an affine transformation of a polytope defined in latent space  $B_\infty(A_e, b_e) \subset \mathbb{R}^{N_C}$ . Here,  $B_\infty(A_e, b_e)$  is defined as the intersection of a unit  $\ell_\infty$ -norm ball and a collection of  $M_C$  linear constraints  $\{\xi \in \mathbb{R}^{N_C} \mid A_e \xi = b_e\}$ .

Formally, a **constrained zonotope** is defined as follows,

$$\mathcal{C} = \{G\xi + c \mid \xi \in B_\infty(A_e, b_e)\} \subset \mathbb{R}^n,$$

where

$$B_\infty(A_e, b_e) = \{\xi \mid \|\xi\|_\infty \leq 1, A_e \xi = b_e\} \subset \mathbb{R}^{N_C},$$

with  $G \in \mathbb{R}^{n \times N_C}$ ,  $c \in \mathbb{R}^n$ ,  $A_e \in \mathbb{R}^{M_C \times N_C}$ , and  $b \in \mathbb{R}^{M_C}$ .

A constrained zonotope provide an alternative and equivalent representation of any convex and compact polytope. Furthermore, a constrained zonotope admits closed-form expressions for several set manipulations that can often be accomplished without invoking any optimization solvers. See [SDGR16] [RK22] [VWD24] for more details.

A **zonotope** is a special class of constrained zonotopes, and are defined as

$$\mathcal{Z} = \{G\xi + c \mid \|\xi\|_\infty \leq 1\} \subset \mathbb{R}^n.$$

In other words, a zonotope is a constrained zonotope with no equality constraints in the latent dimension space. In [ConstrainedZonotope](#), we model zonotopes by having (Ae,be) be empty (n\_equalities is zero).

Constrained zonotope object construction admits **one** of the following combinations (as keyword arguments):

1. dim for an **empty** constrained zonotope of dimension dim,
2. (G, c, Ae, be) for a **constrained zonotope**,
3. (G, c) for a **zonotope**,
4. (lb, ub) for a **zonotope** equivalent to an **axis-aligned cuboid** with appropriate bounds  $\{x \mid lb \leq x \leq ub\}$ , and
5. (c, h) for a **zonotope** equivalent to an **axis-aligned cuboid** centered at c with specified scalar/vector half-sides  $h$ ,  $\{x \mid \forall i \in \{1, 2, \dots, n\}, |x_i - c_i| \leq h_i\}$ .
6. (c=p, G=None) for a **zonotope** equivalent to a **single point** p,
7. P for a **constrained zonotope** equivalent to the [pycvxset.Polytope.Polytope](#) object P,

### Parameters

- **dim** (*int*, *optional*) – Dimension of the empty constrained zonotope. If NOTHING is provided, dim=0 is assumed.

- **c** (*Sequence[float] | numpy.ndarray, optional*) – Affine transformation translation vector. Must be 1D array, and the constrained zonotope dimension is determined by number of elements in c. When c is provided, either (G) or (G, Ae, be) or (h) must be provided additionally. When h is provided, c is the centroid of the resulting zonotope.
- **G** (*Sequence[Sequence[float]] | numpy.ndarray*) – Affine transformation matrix. The vectors are stacked vertically with matching number of rows as c. When G is provided, (c, Ae, be) OR (c) must also be provided. To define a constrained zonotope with a single point, set c to the point AND G to None (do not set (Ae, be) or set them to (None, None)).
- **Ae** (*Sequence[Sequence[float]] | numpy.ndarray*) – Equality coefficient vectors. The vectors are stacked vertically with matching number of columns as G. When Ae is provided, (G, c, be) must also be provided.
- **be** (*Sequence[float] | numpy.ndarray*) – Equality coefficient constants. The constants are expected to be in a 1D numpy array. When be is provided, (G, c, Ae) must also be provided.
- **lb** (*Sequence[float] | numpy.ndarray, optional*) – Lower bounds of the axis-aligned cuboid. Must be 1D array, and the constrained zonotope dimension is determined by number of elements in lb. When lb is provided, ub must also be provided.
- **ub** (*Sequence[float] | numpy.ndarray, optional*) – Upper bounds of the axis-aligned cuboid. Must be 1D array of length as same as lb. When ub is provided, lb must also be provided.
- **h** (*float | Sequence[float] | numpy.ndarray, optional*) – Half-side length of the axis-aligned cuboid. Can be a scalar or a vector of length as same as c. When h is provided, c must also be provided.
- **polytope** (*Polytope, optional*) – Polytope to use to construct constrained zonotope.

#### Raises

- **ValueError** – (G, c) is not compatible.
- **ValueError** – (G, c, Ae, be) is not compatible.
- **ValueError** – (lb, ub) is not valid
- **ValueError** – (c, h) is not valid
- **ValueError** – Provided polytope is not bounded.
- **UserWarning** – When a row with all zeros in Ae and be.

`__init__()` → None

`__init__(*, dim: int)` → None

`__init__(*, G: Sequence[Sequence[float]] | ndarray | None, c: Sequence[float] | ndarray)` → None

`__init__(*, G: Sequence[Sequence[float]] | ndarray | None, c: Sequence[float] | ndarray, Ae: Sequence[Sequence[float]] | ndarray, be: Sequence[float] | ndarray)` → None

`__init__(*, lb: Sequence[float] | ndarray, ub: Sequence[float] | ndarray)` → None

`__init__(*, c: Sequence[float] | ndarray, h: float | Sequence[float] | ndarray)` → None

`__init__(*, polytope: Polytope)` → None

Constructor for ConstrainedZonotope class

## Methods

<code>__init__()</code>	Constructor for ConstrainedZonotope class
<code>affine_map(M)</code>	Multiply a matrix or a scalar with a constrained zonotope
<code>approximate_pontryagin_difference(norm_t, ...)</code>	Approximate Pontryagin difference between a constrained zonotope and an affine transformation of a unit-norm ball.
<code>cartesian_product(sequence_Q)</code>	Generate the Cartesian product of a set $Q$ (or a list of $\mathcal{Q}$ ) with $\mathcal{P}$ .
<code>chebyshev_centering([enable_warning])</code>	Computes a ball with the largest radius that fits within the constrained zonotope.
<code>closest_point(points[, p])</code>	Wrapper for <code>project()</code> to compute the point in the convex set closest to the given point.
<code>containment_constraints(x[, flatten_order])</code>	Get CVXPY constraints for containment of $x$ (a <code>cvxpy.Variable</code> ) in a constrained zonotope.
<code>contains(Q[, verbose, time_limit])</code>	Check containment of a set $Q$ (could be a polytope or a constrained zonotope), or a collection of points $Q \in \mathbb{R}^{n_Q \times \mathcal{P}.dim}$ in the given constrained zonotope.
<code>copy()</code>	Get a copy of the constrained zonotope
<code>distance(points[, p])</code>	Wrapper for <code>project()</code> to compute distance of a point to a convex set.
<code>extreme(eta)</code>	Wrapper for <code>support()</code> to compute the extreme point.
<code>interior_point([point_type, enable_warning])</code>	Compute an interior point of the constrained zonotope.
<code>intersection(Q)</code>	Compute the intersection of constrained zonotope with another constrained zonotope or polytope.
<code>intersection_under_inverse_affine_map(Y, R)</code>	Compute the intersection of constrained zonotope with another constrained zonotope under an inverse affine map
<code>intersection_with_affine_set(Ae, be)</code>	Compute the intersection of a constrained zonotope with an affine set.
<code>intersection_with_halfspaces(A, b)</code>	Compute the intersection of a constrained zonotope with a collection of halfspaces (polyhedron).
<code>inverse_affine_map_under_invertible_matrix(M)</code>	Compute the inverse affine map of a constrained zonotope for a given matrix $M$ .
<code>maximum_volume_inscribing_ellipsoid([...])</code>	Compute the maximum volume ellipsoid that fits within the given constrained zonotope.
<code>minimize(x, objective_to_minimize, cvxpy_args)</code>	Solve a convex program with CVXPY objective subject to containment constraints.
<code>minimum_volume_circumscribing_rectangle([...])</code>	Compute the minimum volume circumscribing rectangle for a set.
<code>minus(Q[, enable_warning])</code>	Implement - operation: Pontryagin difference with a constrained zonotope minuend
<code>plot([method, ax, direction_vectors, ...])</code>	Plot a polytopic approximation of the set.
<code>plus(Q)</code>	Add a point or a set $Q$ to a constrained zonotope (Minkowski sum).
<code>polytopic_inner_approximation([...])</code>	Compute a polytopic inner-approximation of a given set via ray shooting.
<code>polytopic_outer_approximation([...])</code>	Compute a polytopic outer-approximation of a given set via ray shooting.
<code>project(x[, p])</code>	Project a point or a collection of points on to a set.
<code>projection(project_away_dims)</code>	Orthogonal projection of a set $\mathcal{P}$ after removing some user-specified dimensions.
<code>remove_redundancies()</code>	Remove any redundancies in CZ using <code>pycddlib</code> and other geometric properties.

continues on next page

Table 1 – continued from previous page

<code>slice</code> (dims, constants)	Slice a set restricting certain dimensions to constants.
<code>slice_then_projection</code> (dims, constants)	Wrapper for <code>slice()</code> and <code>projection()</code> .
<code>support</code> (eta)	Evaluates the support function and support vector of a set.

## Attributes

<code>Ae</code>	Equality coefficient vectors $Ae$ for the constrained zonotope.
<code>G</code>	Affine transformation matrix $G$ for the constrained zonotope.
<code>He</code>	Equality constraints $He=[Ae, be]$ for the constrained zonotope.
<code>be</code>	Equality constants $be$ for the constrained zonotope.
<code>c</code>	Affine transformation vector $c$ for the constrained zonotope.
<code>cvxpy_args_lp</code>	CVXPY arguments in use when solving a linear program
<code>cvxpy_args_socp</code>	CVXPY arguments in use when solving a second-order cone program
<code>dim</code>	Dimension of the constrained zonotope.
<code>is_bounded</code>	Check if the constrained zonotope is bounded (which is always True)
<code>is_empty</code>	Check if the constrained zonotope is empty
<code>is_full_dimensional</code>	Check if the affine dimension of the constrained zonotope is the same as the constrained zonotope dimension
<code>is_singleton</code>	Check if the constrained zonotope is a singleton
<code>is_zonotope</code>	Check if the constrained zonotope is a zonotope
<code>latent_dim</code>	Latent dimension of the constrained zonotope.
<code>n_equalities</code>	Number of equality constraints used when defining the constrained zonotope.
<code>type_of_set</code>	Return the type of set

`__init__`() → None

`__init__`(\*, *dim*: int) → None

`__init__`(\*, *G*: Sequence[Sequence[float]] | ndarray | None, *c*: Sequence[float] | ndarray) → None

`__init__`(\*, *G*: Sequence[Sequence[float]] | ndarray | None, *c*: Sequence[float] | ndarray, *Ae*: Sequence[Sequence[float]] | ndarray, *be*: Sequence[float] | ndarray) → None

`__init__`(\*, *lb*: Sequence[float] | ndarray, *ub*: Sequence[float] | ndarray) → None

`__init__`(\*, *c*: Sequence[float] | ndarray, *h*: float | Sequence[float] | ndarray) → None

`__init__`(\*, *polytope*: Polytope) → None

Constructor for ConstrainedZonotope class

`__new__`(\*\*kwargs)



## PYCVXSET.CONSTRAINEDZONOTOPE (API DETAILS)

*pycvxset.ConstrainedZonotope*

Constrained zonotope class

**class** pycvxset.ConstrainedZonotope.**ConstrainedZonotope**

Bases: object

Constrained zonotope class

Constrained zonotope defines a polytope in the working dimension  $\mathbb{R}^n$  as an affine transformation of a polytope defined in latent space  $B_\infty(A_e, b_e) \subset \mathbb{R}^{N_C}$ . Here,  $B_\infty(A_e, b_e)$  is defined as the intersection of a unit  $\ell_\infty$ -norm ball and a collection of  $M_C$  linear constraints  $\{\xi \in \mathbb{R}^{N_C} \mid A_e \xi = b_e\}$ .

Formally, a **constrained zonotope** is defined as follows,

$$\mathcal{C} = \{G\xi + c \mid \xi \in B_\infty(A_e, b_e)\} \subset \mathbb{R}^n,$$

where

$$B_\infty(A_e, b_e) = \{\xi \mid \|\xi\|_\infty \leq 1, A_e \xi = b_e\} \subset \mathbb{R}^{N_C},$$

with  $G \in \mathbb{R}^{n \times N_C}$ ,  $c \in \mathbb{R}^n$ ,  $A_e \in \mathbb{R}^{M_C \times N_C}$ , and  $b \in \mathbb{R}^{M_C}$ .

A constrained zonotope provide an alternative and equivalent representation of any convex and compact polytope. Furthermore, a constrained zonotope admits closed-form expressions for several set manipulations that can often be accomplished without invoking any optimization solvers. See [SDGR16] [RK22] [VWD24] for more details.

A **zonotope** is a special class of constrained zonotopes, and are defined as

$$\mathcal{Z} = \{G\xi + c \mid \|\xi\|_\infty \leq 1\} \subset \mathbb{R}^n.$$

In other words, a zonotope is a constrained zonotope with no equality constraints in the latent dimension space. In *ConstrainedZonotope*, we model zonotopes by having (Ae,be) be empty (n\_equalities is zero).

Constrained zonotope object construction admits **one** of the following combinations (as keyword arguments):

1. dim for an **empty** constrained zonotope of dimension dim,
2. (G, c, Ae, be) for a **constrained zonotope**,
3. (G, c) for a **zonotope**,
4. (lb, ub) for a **zonotope** equivalent to an **axis-aligned cuboid** with appropriate bounds  $\{x \mid lb \leq x \leq ub\}$ , and
5. (c, h) for a **zonotope** equivalent to an **axis-aligned cuboid** centered at c with specified scalar/vector half-sides h,  $\{x \mid \forall i \in \{1, 2, \dots, n\}, |x_i - c_i| \leq h_i\}$ .
6. (c=p, G=None) for a **zonotope** equivalent to a **single point** p,
7. P for a **constrained zonotope** equivalent to the *pycvxset.Polytope.Polytope* object P,

## Parameters

- **dim** (*int*, *optional*) – Dimension of the empty constrained zonotope. If NOTHING is provided, dim=0 is assumed.
- **c** (*Sequence[float]* | *numpy.ndarray*, *optional*) – Affine transformation translation vector. Must be 1D array, and the constrained zonotope dimension is determined by number of elements in c. When c is provided, either (G) or (G, Ae, be) or (h) must be provided additionally. When h is provided, c is the centroid of the resulting zonotope.
- **G** (*Sequence[Sequence[float]]* | *numpy.ndarray*) – Affine transformation matrix. The vectors are stacked vertically with matching number of rows as c. When G is provided, (c, Ae, be) OR (c) must also be provided. To define a constrained zonotope with a single point, set c to the point AND G to None (do not set (Ae, be) or set them to (None, None)).
- **Ae** (*Sequence[Sequence[float]]* | *numpy.ndarray*) – Equality coefficient vectors. The vectors are stacked vertically with matching number of columns as G. When Ae is provided, (G, c, be) must also be provided.
- **be** (*Sequence[float]* | *numpy.ndarray*) – Equality coefficient constants. The constants are expected to be in a 1D numpy array. When be is provided, (G, c, Ae) must also be provided.
- **lb** (*Sequence[float]* | *numpy.ndarray*, *optional*) – Lower bounds of the axis-aligned cuboid. Must be 1D array, and the constrained zonotope dimension is determined by number of elements in lb. When lb is provided, ub must also be provided.
- **ub** (*Sequence[float]* | *numpy.ndarray*, *optional*) – Upper bounds of the axis-aligned cuboid. Must be 1D array of length as same as lb. When ub is provided, lb must also be provided.
- **h** (*float* | *Sequence[float]* | *numpy.ndarray*, *optional*) – Half-side length of the axis-aligned cuboid. Can be a scalar or a vector of length as same as c. When h is provided, c must also be provided.
- **polytope** (*Polytope*, *optional*) – Polytope to use to construct constrained zonotope.

## Raises

- **ValueError** – (G, c) is not compatible.
- **ValueError** – (G, c, Ae, be) is not compatible.
- **ValueError** – (lb, ub) is not valid
- **ValueError** – (c, h) is not valid
- **ValueError** – Provided polytope is not bounded.
- **UserWarning** – When a row with all zeros in Ae and be.

`__init__()` → None

`__init__(*, dim: int)` → None

`__init__(*, G: Sequence[Sequence[float]] | ndarray | None, c: Sequence[float] | ndarray)` → None

`__init__(*, G: Sequence[Sequence[float]] | ndarray | None, c: Sequence[float] | ndarray, Ae: Sequence[Sequence[float]] | ndarray, be: Sequence[float] | ndarray)` → None

`__init__(*, lb: Sequence[float] | ndarray, ub: Sequence[float] | ndarray)` → None

`__init__(*, c: Sequence[float] | ndarray, h: float | Sequence[float] | ndarray)` → None

`__init__(*, polytope: Polytope)` → None

Constructor for ConstrainedZonotope class

**affine\_map**(*M*: *int* | *float* | *Sequence*[*float*] | *Sequence*[*Sequence*[*float*]] | *np.ndarray*) → *ConstrainedZonotope*

Multiply a matrix or a scalar with a constrained zonotope

#### Parameters

**M** (*int* | *float* | *Sequence*[*float*] | *Sequence*[*Sequence*[*float*]] | *np.ndarray*) – Matrix (N times self.dim) or a scalar to be multiplied with a constrained zonotope

#### Returns

Constrained zonotope which is the product of M and self. Specifically, given a constrained zonotope  $\mathcal{P}$ , and a matrix  $M \in \mathbb{R}^{m \times P.\text{dim}}$  or scalar  $M$ , then this function returns a constrained zonotope  $\mathcal{R} = \{Mx | x \in \mathcal{P}\}$ .

#### Return type

*ConstrainedZonotope*

#### Notes

This function implements (11) of [SDGR16].

**approximate\_pontryagin\_difference**(*norm\_type*: *int* | *str*, *G\_S*: *Sequence*[*Sequence*[*float*]] | *np.ndarray*, *c\_S*: *Sequence*[*float*] | *np.ndarray*, *method*: *str* = 'inner-least-squares', *enable\_warning*: *bool* = *True*) → *ConstrainedZonotope*

Approximate Pontryagin difference between a constrained zonotope and an affine transformation of a unit-norm ball.

Specifically, we approximate the Pontryagin difference  $\mathcal{P} \ominus \mathcal{S} = \{x \in \mathbb{R}^n \mid x + \mathcal{S} \subseteq \mathcal{P}\}$  between a constrained zonotope  $\mathcal{P}$  and a subtrahend  $\mathcal{S}$ . The subtrahend must be specifically of the form  $\mathcal{S} = \{G_S \xi + c_S \mid \|\xi\|_p \leq 1\}$  for norm\_type  $p \in \{1, 2, \infty\}$ .

- For  $p=1$ , the subtrahend is a convex hull of intervals specified by the columns of  $G_S$  with each interval is symmetric about  $c_S$ .
- For  $p=2$ , the subtrahend is an ellipsoid is characterized by affine transformation  $G_S$  of a unit Euclidean norm ball which is then shifted to  $c_S$ . Here,  $G_S$  is given by `pycvxset.Ellipsoid.Ellipsoid.G`. See `pycvxset.Ellipsoid.Ellipsoid()` to obtain  $G_S$  and  $c_S$  for broader classes of ellipsoids.
- For  $p=\infty$ , the subtrahend is a zonotope characterized by  $(G_S, c_S)$ .

#### Parameters

- **norm\_type** (*int* | *str*) – Norm type.
- **G\_S** (*Sequence*[*Sequence*[*float*]] | *numpy.ndarray*) – Affine transformation matrix (self.dim times N) for scaling the ball
- **c\_S** (*Sequence*[*float*] | *numpy.ndarray*) – Affine transformation vector (self.dim,) for translating the ball
- **method** (*str*, *optional*) – Approximation method to use. Can be one of ['inner-least-squares']. Defaults to 'inner-least-squares'.
- **enable\_warning** (*bool*, *optional*) – Enables the UserWarning. May be turned off if expected. Defaults to True.

#### Raises

- **ValueError** – Norm type is not in [1, 2, 'inf']
- **ValueError** – Mismatch in dimension (no. of columns of G\_S, length of c\_S, and self.dim should match)

### Returns

Approximated Pontryagin difference between self and the affine transformation of the unit-norm ball.

### Return type

*ConstrainedZonotope*

### Notes

For method ‘inner-least-squares’, this function implements Table 1 of [VWD24].

**cartesian\_product**(*sequence\_Q*: *Polytope* | *ConstrainedZonotope* | *Sequence*[*Polytope* | *ConstrainedZonotope*]) → *ConstrainedZonotope*

Generate the Cartesian product of a set  $Q$  (or a list of  $\mathcal{Q}$ ) with  $\mathcal{P}$ .

### Parameters

**sequence\_Q** (*Polytope* | *ConstrainedZonotope* | *Sequence*[*Polytope* | *ConstrainedZonotope*]) – List of sets to take Cartesian product with

### Returns

Cartesian product of self and all sets in *sequence\_Q*

### Return type

*ConstrainedZonotope*

**chebyshev\_centering**(*enable\_warning*: *bool* = *True*) → tuple[np.ndarray, float]

Computes a ball with the largest radius that fits within the constrained zonotope. The ball’s center is known as the Chebyshev center, and its radius is the Chebyshev radius.

### Parameters

**enable\_warning** (*bool*, *optional*) – Enables the UserWarning. May be turned off if expected. Defaults to *True*.

### Raises

- **ValueError** – When the constrained zonotope is not full-dimensional
- **ValueError** – When the constrained zonotope is empty
- **NotImplementedError** – Unable to solve the linear program using CVXPY

### Returns

A tuple with two items

1. center (numpy.ndarray): Approximate Chebyshev radius of the constrained zonotope
2. radius (float): Approximate Chebyshev radius of the constrained zonotope

### Return type

tuple

### Notes

Unlike *pycvxset.Polytope.Polytope.chebyshev\_centering()*, this function computes an approximate Chebyshev center and radius. Specifically, it guarantees that a ball of the computed radius, centered at the computed center is contained in the constrained zonotope. However, it does not guarantee that the computed radius is the radius of the largest possible ball contained in the given constrained zonotope. For more details, see [VWS24].

**closest\_point**(*points*: *Sequence*[*Sequence*[float]] | *np.ndarray*, *p*: *int* | *str* = 2) → *np.ndarray*

Wrapper for *project()* to compute the point in the convex set closest to the given point.

### Parameters

- **points** (*Sequence*[*Sequence*[float]] | *np.ndarray*) – Points to project. Matrix (N times self.dim), where each row is a point.

- **p** (*str* / *int*) – Norm-type. It can be 1, 2, or ‘inf’. Defaults to 2.

#### Returns

Projection of points to the set as a 2D numpy.ndarray. These arrays have as many rows as points.

#### Return type

numpy.ndarray

#### Notes

For more detailed description, see documentation for [project\(\)](#) function.

**containment\_constraints** (*x*: *cvxpy.Variable*, *flatten\_order*: *Literal['F', 'C'] = 'F'*) → *tuple*[*list*[*cvxpy.Constraint*], *cvxpy.Variable* | *None*]

Get CVXPY constraints for containment of *x* (a *cvxpy.Variable*) in a constrained zonotope.

#### Parameters

- **x** (*cvxpy.Variable*) – CVXPY variable to be optimized.
- **flatten\_order** (*Literal["F", "C"]*) – Order to use for flatten (choose between “F”, “C”). Defaults to “F”, which implements column-major flatten. In 2D, column-major flatten results in stacking rows horizontally to achieve a single horizontal row.

#### Raises

**ValueError** – When constrained zonotope is empty

#### Returns

A tuple with two items:

1. *constraint\_list* (*list*): CVXPY constraints for the containment of *x* in the constrained zonotope.
2. *xi* (*cvxpy.Variable* | *None*): CVXPY variable representing the latent dimension variable. It is *None*, when the constrained zonotope is a single point.

#### Return type

*tuple*

#### Notes

This function imports CVXPY locally to avoid overhead.

**contains** (*Q*: *Sequence[float]* | *Sequence[Sequence[float]]* | *np.ndarray* | [Polytope](#) | [ConstrainedZonotope](#), *verbose*: *bool = False*, *time\_limit*: *float = 60*) → *bool* | *np.ndarray*

Check containment of a set *Q* (could be a polytope or a constrained zonotope), or a collection of points  $Q \in \mathbb{R}^{n_Q \times \mathcal{P}.dim}$  in the given constrained zonotope.

#### Parameters

- **Q** (*Sequence[float]* | *Sequence[Sequence[float]]* | *np.ndarray* | [Polytope](#) | [ConstrainedZonotope](#)) – Polytope object, [ConstrainedZonotope](#) object, or a collection of points to be tested for containment within the constrained zonotope. When providing a collection of points, *Q* is a matrix (N times self.dim) with each row is a point.
- **verbose** (*bool*, *optional*) – Verbosity flag to provide cvxpy when solving the MIQP related to checking containment of a constrained zonotope within another constrained zonotope. Defaults to False.
- **time\_limit** (*float*, *optional*) – Time limit in seconds for GUROBI solver when solving the MIQP related to checking containment of a constrained zonotope within another constrained zonotope. Set *time\_limit* to *np.inf* if no limit is desired. Defaults to 60s (see [constants.py](#)).

### Raises

- **ValueError** – Dimension mismatch between  $Q$  and the constrained zonotope
- **ValueError** –  $Q$  is Ellipsoid
- **ValueError** – Unable to perform containment check between two constrained zonotopes (including `time_limit` issues)
- **NotImplementedError** – GUROBI is not installed when checking containment of two constrained zonotopes
- **NotImplementedError** – Failed to check containment between two constrained zonotopes, due to an unhandled status

### Returns

Boolean corresponding to  $Q \subseteq \mathcal{P}$  or  $Q \in \mathcal{P}$ .

### Return type

bool | numpy.ndarray([bool])

### Notes

- We use  $\mathcal{P}$  to denote the constrained zonotope characterized by `self`.
- When  $Q$  is a constrained zonotope, a bool is returned which is True if and only if  $Q \subseteq \mathcal{P}$ . This function uses the non-convex programming capabilities of GUROBI (via CVXPY) to check containment between two constrained zonotopes.
- When  $Q$  is a polytope, a bool is returned which is True if and only if  $Q \subseteq \mathcal{P}$ .
  - When  $Q$  is in V-Rep, the containment problem simplifies to checking if all vertices of  $Q$  are contained  $\mathcal{P}$ .
  - When  $Q$  is in H-Rep, this function converts  $Q$  into a constrained zonotope, and then checks for containment.
- When  $Q$  is a single  $n$ -dimensional point, a bool is returned which is True if and only if  $Q \in \mathcal{P}$ . This function uses `distance()` to check containment of a point in a constrained zonotope.
- When  $Q$  is a collection of  $n$ -dimensional points ( $Q$  is a matrix with each row describing a point), an array of is returned where each element is True if and only if  $Q_i \in \mathcal{P}$ . This function uses `distance()` to check containment of a point in a constrained zonotope.

### Warning

This function requires `gurobipy` when checking if the `ConstrainedZonotope` object represented by `self` contains a `ConstrainedZonotope` object  $Q$  or `pycvxset.Polytope.Polytope` object  $Q$  in H-Rep.

`copy()` → `ConstrainedZonotope`

Get a copy of the constrained zonotope

`distance(points: Sequence[Sequence[float]] | np.ndarray, p: int | str = 2) → np.ndarray`

Wrapper for `project()` to compute distance of a point to a convex set.

### Parameters

- **points** (`Sequence[Sequence[float]]` | `np.ndarray`) – Points to project. Matrix ( $N$  times `self.dim`), where each row is a point.
- **p** (`int` | `str`) – Norm-type. It can be 1, 2, or 'inf'. Defaults to 2.

**Returns**

Distance of points to the set as a 1D numpy.ndarray. These arrays have as many rows as points.

**Return type**

numpy.ndarray

**Notes**

For more detailed description, see documentation for [project\(\)](#) function.

**extreme**(*eta*: Sequence[Sequence[float]] | np.ndarray) → np.ndarray

Wrapper for [support\(\)](#) to compute the extreme point.

**Parameters**

**eta** (Sequence[Sequence[float]] | np.ndarray) – Support directions. Matrix (N times self.dim), where each row is a support direction.

**Returns**

Support vector evaluation(s) as a 2D numpy.ndarray. The array has as many rows as eta.

**Return type**

numpy.ndarray

**Notes**

For more detailed description, see documentation for [support\(\)](#) function.

**interior\_point**(*point\_type*: str = 'mvie', *enable\_warning*: bool = True) → np.ndarray

Compute an interior point of the constrained zonotope.

**Parameters**

- **point\_type** (*str*, *optional*) – Type of interior point. Valid strings: {'mvie', 'chebyshev'}. Defaults to 'mvie'.
- **enable\_warning** (*bool*, *optional*) – Enables the UserWarning. May be turned off if expected. Defaults to True.

**Returns**

A point that lies in the (relative) interior of the constrained zonotope.

**Return type**

numpy.ndarray

**Notes**

This function is a wrapper for existing centering methods in ConstrainedZonotope class — `maximum_volume_inscribing_ellipsoid`, `chebyshev_centering`. For the case where self is a zonotope, it returns `c`.

**intersection**(*Q*: ConstrainedZonotope | Polytope) → ConstrainedZonotope

Compute the intersection of constrained zonotope with another constrained zonotope or polytope.

**Parameters**

**Q** (ConstrainedZonotope | Polytope) – Set to intersect with

**Raises**

**TypeError** – When Q is neither a ConstrainedZonotope object or a Polytope object

**Returns**

Intersection of self with Q

**Return type**

ConstrainedZonotope

## Notes

- When  $Q$  is a constrained zonotope, this function uses `intersection_under_inverse_affine_map()` with  $R$  set to identity matrix.
- When  $Q$  is a polytope in H-Rep, this function uses `intersection_with_halfspaces()` and `intersection_with_affine_set()`.
- When  $Q$  is a polytope in V-Rep, this function converts  $Q$  into a constrained zonotope to avoid a halfspace enumeration.

**intersection\_under\_inverse\_affine\_map**( $Y$ : `ConstrainedZonotope`,  $R$ : `Sequence[Sequence[float]]` | `np.ndarray`)  $\rightarrow$  `ConstrainedZonotope`

Compute the intersection of constrained zonotope with another constrained zonotope under an inverse affine map

### Parameters

- **Y** (`ConstrainedZonotope`) – Set to intersect with
- **R** (`Sequence[Sequence[float]]` | `numpy.ndarray`) – Matrix ( $Y.\text{dim}$  times  $\text{self}.\text{dim}$ ) for the inverse-affine map.

### Raises

- **ValueError** – When  $Y$  is not a `ConstrainedZonotope`
- **ValueError** – When  $R$  is not of correct dimension

### Returns

Intersection of self with  $Y$  under an inverse affine map  $R$ . Specifically, given constrained zonotopes  $\mathcal{P}$  (self) and  $\mathcal{Y}$ , and a matrix  $R$ , we compute the set  $\mathcal{P} \cap_R \mathcal{Y} = \{x \in \mathcal{P} | Rx \in \mathcal{Y}\}$ . When  $R$  is invertible,  $\mathcal{P} \cap_R \mathcal{Y} = (R^{-1}\mathcal{P}) \cap \mathcal{Y}$ .

### Return type

`ConstrainedZonotope`

## Notes

This function implements (13) of [SDGR16]. This function does not require  $R$  to be invertible.

**intersection\_with\_affine\_set**( $A_e$ : `Sequence[Sequence[float]]` | `np.ndarray`,  $b_e$ : `Sequence[float]` | `np.ndarray`)  $\rightarrow$  `ConstrainedZonotope`

Compute the intersection of a constrained zonotope with an affine set.

### Parameters

- **Ae** (`Sequence[Sequence[float]]` | `numpy.ndarray`) – Equality coefficient matrix ( $N$  times  $\text{self}.\text{dim}$ ) that define the affine set  $\{x | A_e x = b_e\}$ .
- **be** (`Sequence[float]` | `numpy.ndarray`) – Equality constant vector ( $N$ ,) that define the affine set  $\{x | A_e x = b_e\}$ .

### Raises

**ValueError** – When the number of columns in  $A_e$  is different from  $\text{self}.\text{dim}$

### Returns

The intersection of a constrained zonotope with the affine set.

### Return type

`ConstrainedZonotope`



## Notes

This function implements imposes the constraints  $\{A_e x = b_e\}$  as constraints in the latent dimension of the constrained zonotope —  $A_e(G\xi + c) = b_e$  for every feasible  $\xi$ .

**intersection\_with\_halfspaces**(*A*: Sequence[Sequence[float]] | np.ndarray, *b*: Sequence[float] | np.ndarray) → *ConstrainedZonotope*

Compute the intersection of a constrained zonotope with a collection of halfspaces (polyhedron).

### Parameters

- **A** (Sequence[Sequence[float]] | numpy.ndarray) – Inequality coefficient matrix (N times self.dim) that define the polyhedron  $\{x | Ax \leq b\}$ .
- **b** (Sequence[float] | numpy.ndarray) – Inequality constant vector (N,) that define the polyhedron  $\{x | Ax \leq b\}$ .

### Returns

The intersection of a constrained zonotope with a collection of halfspaces.

### Return type

*ConstrainedZonotope*

## Notes

This function implements (10) of [RK22] for each halfspace. We skip redundant inequalities when encountered and we return empty set when intersection yields an empty set.

**inverse\_affine\_map\_under\_invertible\_matrix**(*M*: Sequence[Sequence[float]] | np.ndarray) → *ConstrainedZonotope*

Compute the inverse affine map of a constrained zonotope for a given matrix M.

### Parameters

**M** (Sequence[Sequence[float]] | numpy.ndarray) – Square invertible matrix of dimension self.dim

### Raises

- **ValueError** – M is not convertible into a 2D float array
- **ValueError** – M is not square
- **ValueError** – M is not invertible

### Returns

Inverse affine map of self under M. Specifically, Given a constrained zonotope  $\mathcal{P}$  and an invertible self.dim-dimensional matrix  $M \in \mathbb{R}^{\mathcal{P}.dim \times \mathcal{P}.dim}$ , this function computes the constrained zonotope  $\mathcal{R} = M^{-1}\mathcal{P}$ .

### Return type

*ConstrainedZonotope*

## Notes

This function is a wrapper for [affine\\_map\(\)](#). We require M to be invertible in order to ensure that the resulting set is representable as a constrained zonotope.

**maximum\_volume\_inscribing\_ellipsoid**(*enable\_warning*: bool = True) → tuple[np.ndarray, np.ndarray, np.ndarray]

Compute the maximum volume ellipsoid that fits within the given constrained zonotope.

### Parameters

**enable\_warning** (bool, optional) – Enables the UserWarning. May be turned off if expected. Defaults to True.

### Raises

- **ValueError** – When the constrained zonotope is not full-dimensional
- **ValueError** – When the constrained zonotope is empty
- **NotImplementedError** – Unable to solve the convex program using CVXPY

#### Returns

##### A tuple with three items:

1. `center` (numpy.ndarray): Approximate maximum volume inscribed ellipsoid's center
2. `shape_matrix` (numpy.ndarray): Approximate maximum volume inscribed ellipsoid's shape matrix
3. `sqrt_shape_matrix` (numpy.ndarray): Approximate maximum volume inscribed ellipsoid's square root of shape matrix.

#### Return type

tuple

#### Notes

Unlike `pycvxset.Polytope.Polytope.maximum_volume_inscribing_ellipsoid()`, this function computes an approximate maximum volume inscribed ellipsoid. Specifically, it guarantees that the computed ellipsoid is contained in the constrained zonotope. However, it does not guarantee that the computed ellipsoid is the largest possible ellipsoid (in terms of volume) contained in the given constrained zonotope. For more details, see [VWS24].

**minimize**(*x*: `cvxpy.Variable`, *objective\_to\_minimize*: `cvxpy.Expression`, *cvxpy\_args*: `dict[str, Any]`, *task\_str*: `str = ""`) → `tuple[np.ndarray, float, str]`

Solve a convex program with CVXPY objective subject to containment constraints.

#### Parameters

- **x** (`cvxpy.Variable`) – CVXPY variable to be optimized
- **objective\_to\_minimize** (`cvxpy.Expression`) – CVXPY expression to be minimized
- **cvxpy\_args** (`dict`) – CVXPY arguments to be passed to the solver
- **task\_str** (`str`, *optional*) – Task string to be used in error messages. Defaults to “.”.

#### Raises

**NotImplementedError** – Unable to solve problem using CVXPY

#### Returns

##### A tuple with three items:

1. `x.value` (numpy.ndarray): Optimal value of `x`. `np.nan * np.ones((self.dim,))` if the problem is not solved.
2. `problem.value` (float): Optimal value of the convex program. `np.inf` if the problem is infeasible, `-np.inf` if problem is unbounded, and finite otherwise.
3. `problem_status` (`str`): Status of the problem

#### Return type

tuple

## Notes

This function uses `containment_constraints()` to obtain the list of CVXPY expressions that form the containment constraints on  $x$ .

### Warning

Please pay attention to the `NotImplementedError` generated by this function. It may be possible to get CVXPY to solve the same problem by switching the solver. For example, consider the following code block.

```
from pycvxset import Polytope
P = Polytope(A=[[1, 1], [-1, -1]], b=[1, 1])
P.cvxpy_args_lp = {'solver': 'CLARABEL'} # Default solver used in
↳pycvxset
try:
    print('Is polytope bounded?', P.is_bounded)
except NotImplementedError as err:
    print(str(err))
P.cvxpy_args_lp = {'solver': 'OSQP'}
print('Is polytope bounded?', P.is_bounded)
```

This code block produces the following output:

```
Unable to solve the task (support function evaluation of the set at eta =
↳[-0. -1.]). CVXPY returned error:
Solver 'CLARABEL' failed. Try another solver, or solve with verbose=True
↳for more information.
```

```
Is polytope bounded? False
```

**minimum\_volume\_circumscribing\_rectangle()** → tuple[np.ndarray, np.ndarray]

Compute the minimum volume circumscribing rectangle for a set.

#### Raises

**ValueError** – Solver error or set is empty!

#### Returns

**A tuple of two elements**

- lb (numpy.ndarray): Lower bound  $l$  on the set,  $\mathcal{P} \subseteq \{l\} \oplus \mathbb{R}_{\geq 0}$ .
- ub (numpy.ndarray): Upper bound  $u$  on the set,  $\mathcal{P} \subseteq \{u\} \oplus (-\mathbb{R}_{\geq 0})$ .

#### Return type

tuple

## Notes

This function computes the lower/upper bound by an element-wise support computation ( $2n$  linear programs), where  $n$  is `attr:self.dim`. To reuse the `support()` function for the lower bound computation, we solve the optimization for each  $i \in \{1, 2, \dots, n\}$ ,

$$\inf_{x \in \mathcal{P}} e_i^\top x = - \sup_{x \in \mathcal{P}} -e_i^\top x = -\rho_{\mathcal{P}}(-e_i),$$

where  $e_i \in \mathbb{R}^n$  denotes the standard coordinate vector, and  $\rho_{\mathcal{P}}$  is the support function of  $\mathcal{P}$ .

**minus**( $Q$ : Sequence[float] | np.ndarray | Ellipsoid | ConstrainedZonotope, enable\_warning: bool = True)  
→ ConstrainedZonotope

Implement - operation: Pontryagin difference with a constrained zonotope minuend

When  $Q$  is an ellipsoid or a zonotope, minus returns an inner-approximation of the set corresponding to the Pontryagin difference of a constrained zonotope and  $Q$ . When  $Q$  is a point or a singleton set, an exact set corresponding to the translation by  $-Q$  or  $-Q.c$  is returned.

#### Parameters

- **Q** (*Sequence[float] | numpy.ndarray | Ellipsoid | ConstrainedZonotope*) – Point/set to use as subtrahend in the Pontryagin difference.
- **enable\_warning** (*bool, optional*) – Enables the UserWarning. May be turned off if expected. Defaults to True.

#### Raises

- **TypeError** – When  $Q$  is not one of the following — convertible into a 1D numpy array, or an ellipsoid, or a zonotope.
- **ValueError** – When  $Q$  has a dimension mismatch with self.
- **UserWarning** – When using with  $Q$  that is a set, warn that an inner-approximation is returned.

#### Returns

Pontryagin difference of self and  $Q$

#### Return type

*ConstrainedZonotope*

#### Notes

This function uses `approximate_pontryagin_difference()` when  $Q$  is a set and uses `plus()` when  $Q$  is a point.

**plot**(*method: str = 'inner', ax: Axes | Axes3D | None | None = None, direction\_vectors: Sequence[Sequence[float]] | np.ndarray | None = None, n\_vertices: int | None = None, n\_halfspaces: int | None = None, patch\_args: dict[str, Any] | None = None, vertex\_args: dict[str, Any] | None = None, center\_args: dict[str, Any] | None = None, autoscale\_enable: bool = True, decimal\_precision: int = 3, enable\_warning: bool = True*) → tuple[Any, ...]

Plot a polytopic approximation of the set.

#### Parameters

- **method** (*str, optional*) – Type of polytopic approximation to use. Can be [“inner” or “outer”]. Defaults to “inner”.
- **ax** (*Axes | Axes3D | None, optional*) – Axis on which the patch is to be plotted
- **direction\_vectors** (*Sequence[Sequence[float]] | np.ndarray, optional*) – Directions to use when performing ray shooting. Matrix (N times self.dim) for some  $N \geq 1$ . Defaults to None, in which case we use `pycvxset.common.spread_points_on_a_unit_sphere()` to compute the direction vectors.
- **n\_vertices** (*int, optional*) – Number of vertices to use when computing the polytopic inner-approximation. Ignored if method is “outer” or when direction\_vectors are provided. More than n\_vertices may be used in some cases (see notes). Defaults to None.
- **n\_halfspaces** (*int, optional*) – Number of halfspaces to use when computing the polytopic outer-approximation. Ignored if method is “outer” or when direction\_vectors are provided. More than n\_halfspaces may be used in some cases (see notes). Defaults to None.
- **patch\_args** (*dict, optional*) – Arguments to pass for plotting faces and edges. See `pycvxset.Polytope.Polytope.plot()` for more details. Defaults to None.

- **vertex\_args** (*dict, optional*) – Arguments to pass for plotting vertices. See `pycvxset.Polytope.Polytope.plot()` for more details. Defaults to None.
- **center\_args** (*dict, optional*) – For ellipsoidal set, arguments to pass to scatter plot for the center. If a label is desired, pass it in center\_args.
- **autoscale\_enable** (*bool, optional*) – When set to True, matplotlib adjusts axes to view full polytope. See `pycvxset.Polytope.Polytope.plot()` for more details. Defaults to True.
- **decimal\_precision** (*int, optional*) – When plotting a 3D polytope that is in V-Rep and not in H-Rep, we round vertex to the specified precision to avoid numerical issues. Defaults to PLOTTING\_DECIMAL\_PRECISION\_CDD specified in `pycvxset.common.constants`.
- **enable\_warning** (*bool, optional*) – Enables the UserWarning. May be turned off if expected. Defaults to True.

#### Returns

See `pycvxset.Polytope.Polytope.plot()` for details.

#### Return type

tuple

#### Notes

This function is a wrapper for `polytopic_inner_approximation()` and `polytopic_outer_approximation()` for more details in polytope construction.

**plus**(*Q: Sequence[float] | np.ndarray | Polytope | ConstrainedZonotope*) → *ConstrainedZonotope*

Add a point or a set Q to a constrained zonotope (Minkowski sum).

#### Parameters

**Q** (*Sequence[float] | np.ndarray | Polytope | ConstrainedZonotope*) – The point or set to add

#### Raises

- **TypeError** – When Q is not one of the following — convertible into a 1D numpy array or a constrained zonotope.
- **ValueError** – When Q has a dimension mismatch with self.
- **UserWarning** – When using with Q that is a set, warn that an inner-approximation is returned.

#### Returns

Minkowski sum of self and Q.

#### Return type

*ConstrainedZonotope*

#### Notes

Given a constrained zonotope  $\mathcal{P}$  and a set  $Q$ , this function computes the Minkowski sum of Q and the constrained zonotope, defined as  $\mathcal{R} = \{x + q | x \in \mathcal{P}, q \in Q\}$ . On the other hand, when Q is a point, this function computes the constrained zonotope  $\mathcal{R} = \{x + Q | x \in \mathcal{P}\}$ .

This function implements (12) of [SDGR16] when Q is a constrained zonotope. When Q is a Polytope, this function converts it into a constrained zonotope, and then uses (12) of [SDGR16].

**polytopic\_inner\_approximation**(*direction\_vectors: Sequence[Sequence[float]] | np.ndarray | None = None, n\_vertices: int | None = None, verbose: bool = False, enable\_warning: bool = True*) → *Polytope*

Compute a polytopic inner-approximation of a given set via ray shooting.

#### Parameters

- **cvx\_set** ([ConstrainedZonotope](#) / [Ellipsoid](#) / [Polytope](#)) – Set to be approximated,
- **direction\_vectors** ([Sequence\[Sequence\[float\]\]](#) | [np.ndarray](#), *optional*) – Directions to use when performing ray shooting. Matrix ( $N$  times  $\text{self.dim}$ ) for some  $N \geq 1$ . Defaults to None.
- **n\_vertices** (*int*, *optional*) – Number of vertices to be used for the inner-approximation. `n_vertices` is overridden whenever `direction_vectors` are provided. Defaults to None.
- **verbose** (*bool*, *optional*) – If true, [pycvxset.common.spread\\_points\\_on\\_a\\_unit\\_sphere\(\)](#) is passed with `verbose`. Defaults to False.
- **enable\_warning** (*bool*, *optional*) – Enables the UserWarning. May be turned off if expected. Defaults to True.

#### Returns

Polytopic inner-approximation in V-Rep of a given set with `n_vertices` no smaller than user-provided `n_vertices`.

#### Return type

[Polytope](#)

#### Notes

We compute the polytope using [extreme\(\)](#) evaluated along the direction vectors computed by [pycvxset.common.spread\\_points\\_on\\_a\\_unit\\_sphere\(\)](#). When `direction_vectors` is None and `n_vertices` is None, we select `n_vertices = 2self.dim + 2self.dimSPOAUS_DIRECTIONS_PER_QUADRANT` (as defined in [pycvxset.common.constants\(\)](#)). [BV04] The function also uses [pycvxset.common.make\\_aspect\\_ratio\\_equal\(\)](#) to account for possibly non-symmetric sets.

**polytopic\_outer\_approximation**(*direction\_vectors: Sequence[Sequence[float]] | np.ndarray | None = None, n\_halfspaces: int | None = None, verbose: bool = False, enable\_warning: bool = True*) → [Polytope](#)

Compute a polytopic outer-approximation of a given set via ray shooting.

#### Parameters

- **cvx\_set** ([ConstrainedZonotope](#) / [Ellipsoid](#) / [Polytope](#)) – Set to be approximated,
- **direction\_vectors** ([Sequence\[Sequence\[float\]\]](#) | [np.ndarray](#), *optional*) – Directions to use when performing ray shooting. Matrix ( $N$  times  $\text{self.dim}$ ) for some  $N \geq 1$ . Defaults to None.
- **n\_halfspaces** (*int*, *optional*) – Number of halfspaces to be used for the inner-approximation. `n_vertices` is overridden whenever `direction_vectors` are provided. Defaults to None.
- **verbose** (*bool*, *optional*) – If true, [pycvxset.common.spread\\_points\\_on\\_a\\_unit\\_sphere\(\)](#) is passed with `verbose`. Defaults to False.
- **enable\_warning** (*bool*, *optional*) – Enables the UserWarning. May be turned off if expected. Defaults to True.

#### Returns

Polytopic outer-approximation in H-Rep of a given set with `n_halfspaces` no smaller than user-provided `n_vertices`.

#### Return type

[Polytope](#)

## Notes

We compute the polytope using `support()` evaluated along the direction vectors computed by `pycvxset.common.spread_points_on_a_unit_sphere()`. When `direction_vectors` is `None` and `n_halfspaces` is `None`, we select `n_halfspaces = 2self.dim + 2self.dim*SPOAUS_DIRECTIONS_PER_QUADRANT` (as defined in `pycvxset.common.constants`). [BV04]

**project**(*x*: Sequence[float] | Sequence[Sequence[float]] | ndarray, *p*: int | str = 2) → tuple[ndarray, ndarray]

Project a point or a collection of points on to a set.

Given a set  $\mathcal{P}$  and a test point  $y \in \mathbb{R}^{\mathcal{P}.dim}$ , this function solves a convex program,

$$\begin{aligned} &\text{minimize} && \|x - y\|_p \\ &\text{subject to} && x \in \mathcal{P} \end{aligned}$$

## Parameters

- **points** (Sequence[Sequence[float]] | np.ndarray) – Points to project (N times self.dim) with each row as a point.
- **p** (str / int) – Norm-type. It can be 1, 2, or ‘inf’. Defaults to 2, which is the Euclidean norm.

## Raises

- **ValueError** – Set is empty
- **ValueError** – Dimension mismatch — no. of columns in points is different from self.dim.
- **ValueError** – Points is not convertible into a 2D array
- **NotImplementedError** – Unable to solve problem using CVXPY

## Returns

A tuple with two items:

1. `projected_point` (numpy.ndarray): Projection point(s) as a 2D numpy.ndarray. Matrix (N times self.dim), where each row is a projection of the point in points to the set  $\mathcal{P}$ .
2. `distance` (numpy.ndarray): Distance(s) as a 1D numpy.ndarray. Vector (N,), where each row is a projection of the point in points to the set  $\mathcal{P}$ .

## Return type

tuple

## Notes

Given a constrained zonotope  $\mathcal{P}$  and a test point  $y \in \mathbb{R}^{\mathcal{P}.dim}$ , this function solves a convex program with decision variables  $x \in \mathbb{R}^{\mathcal{P}.dim}$  and  $\xi \in \mathbb{R}^{\mathcal{P}.latent\_dim}$ ,

$$\begin{aligned} &\text{minimize} && \|x - y\|_p \\ &\text{subject to} && x = G_P \xi + c_P \\ & && A_P \xi = b_P \\ & && \|\xi\|_\infty \leq 1 \end{aligned}$$

**projection**(*project\_away\_dims*: *int* | *Sequence[int]* | *ndarray*) → *ConstrainedZonotope*

Orthogonal projection of a set  $\mathcal{P}$  after removing some user-specified dimensions.

$$\mathcal{R} = \{r \in \mathbb{R}^m \mid \exists v \in \mathbb{R}^{n-m}, \text{Lift}(r, v) \in \mathcal{P}\}$$

Here,  $m = \mathcal{P}.\text{dim} - \text{length}(\text{project\_away\_dim})$ , and  $\text{Lift}(r, v)$  lifts (“undo”s the projection) using the appropriate components of  $v$ . This function uses [affine\\_map\(\)](#) to implement the projection by designing an appropriate affine map  $M \in \{0, 1\}^{m \times \mathcal{P}.\text{dim}}$  with each row of  $M$  corresponding to some standard axis vector  $e_i \in \mathbb{R}^m$ .

**Parameters**

**project\_away\_dims** (*Sequence[int]* | *np.ndarray*) – Dimensions to projected away in integer interval  $[0, 1, \dots, n - 1]$ .

**Raises**

**ValueError** – When *project\_away\_dims* are not in the integer interval | All dimensions are projected away

**Returns**

Set obtained via projection.

**Return type**

object

**Returns**

m-dimensional set obtained via projection.

**Return type**

*ConstrainedZonotope*

**remove\_redundancies**() → None

Remove any redundancies in CZ using pycddlib and other geometric properties.

Updates full-dimensional flag when empty

Updates empty flag when empty OR non-empty zonotope OR when latent set is non-empty

**slice**(*dims*: *int* | *Sequence[int]* | *ndarray*, *constants*: *float* | *Sequence[float]* | *ndarray*) →

*ConstrainedZonotope*

Slice a set restricting certain dimensions to constants.

This function uses [intersection\\_with\\_affine\\_set\(\)](#) to implement the slicing by designing an appropriate affine set from *dims* and *constants*.

**Parameters**

- **dims** (*Sequence[int]* | *np.ndarray*) – List of dims to restrict to a constant in the integer interval  $[0, 1, \dots, n - 1]$ .

- **constants** (*float* | *Sequence[float]* | *np.ndarray*) – List of constants

**Raises**

- **ValueError** – *dims* has entries beyond *n*
- **ValueError** – *dims* and *constants* are not 1D arrays of same size

**Returns**

Sliced set.

**Return type**

object

**Returns**

Constrained zonotope that has been sliced at the specified dimensions.

**Return type**

*ConstrainedZonotope*



**slice\_then\_projection**(*dims*: *int* | *Sequence*[*int*] | *ndarray*, *constants*: *float* | *Sequence*[*float*] | *ndarray*) → *ConstrainedZonotope*

Wrapper for *slice()* and *projection()*.

The function first restricts a set at certain dimensions to constants, and then projects away those dimensions. Useful for visual inspection of higher dimensional sets.

#### Parameters

- **dims** (*Sequence*[*int*] | *np.ndarray*) – List of dims to restrict to a constant in the integer interval  $[0, 1, \dots, \text{dim} - 1]$ , and then project away.
- **constants** (*float* | *Sequence*[*float*] | *np.ndarray*) – List of constants

#### Raises

- **ValueError** – *dims* has entries beyond *n*
- **ValueError** – *dims* and *constants* are not 1D arrays of same size
- **ValueError** – When *dims* are not in the integer interval | All dimensions are projected away

#### Returns

Sliced then projected set.

#### Return type

object

#### Returns

*m*-dimensional set obtained via projection after slicing.

#### Return type

*ConstrainedZonotope*

**support**(*eta*: *Sequence*[*float*] | *Sequence*[*Sequence*[*float*]] | *ndarray*) → *tuple*[*ndarray*, *ndarray*]

Evaluates the support function and support vector of a set.

The support function of a set  $\mathcal{P}$  is defined as  $\rho_{\mathcal{P}}(\eta) = \max_{x \in \mathcal{P}} \eta^{\top} x$ . The support vector of a set  $\mathcal{P}$  is defined as  $\nu_{\mathcal{P}}(\eta) = \arg \max_{x \in \mathcal{P}} \eta^{\top} x$ .

#### Parameters

**eta** (*Sequence*[*float*] | *Sequence*[*Sequence*[*float*]] | *np.ndarray*) – Support directions. Matrix (*N* times *self.dim*), where each row is a support direction.

#### Raises

- **ValueError** – Set is empty
- **ValueError** – Mismatch in *eta* dimension
- **ValueError** – *eta* is not convertible into a 2D array
- **NotImplementedError** – Unable to solve problem using CVXPY

#### Returns

A tuple with two items:

1. *support\_function\_evaluations* (*numpy.ndarray*): Support function evaluation(s) as a 2D *numpy.ndarray*. Vector (*N*), with as many rows as *eta*.
2. *support\_vectors* (*numpy.ndarray*): Support vectors as a 2D *numpy.ndarray*. Matrix *N* x *self.dim* with as many rows as *eta*.

#### Return type

tuple

## Notes

Given a constrained zonotope  $\mathcal{P}$  and a support direction  $\eta \in \mathbb{R}^{\mathcal{P}.\text{dim}}$ , this function solves a convex program with decision variables  $x \in \mathbb{R}^{\mathcal{P}.\text{dim}}$  and  $\xi \in \mathbb{R}^{\mathcal{P}.\text{latent\_dim}}$ ,

$$\begin{aligned} &\text{maximize} && \eta^\top x \\ &\text{subject to} && x = G_P \xi + c_P \\ & && A_P \xi = b_P \\ & && \|\xi\|_\infty \leq 1 \end{aligned}$$

### property Ae: ndarray

Equality coefficient vectors Ae for the constrained zonotope.

#### Returns

Equality coefficient vectors Ae for the constrained zonotope. Ae is `np.empty((0, self.latent_dim))` for a zonotope.

#### Return type

`numpy.ndarray`

### property G: ndarray

Affine transformation matrix G for the constrained zonotope.

#### Returns

Affine transformation matrix G.

#### Return type

`numpy.ndarray`

### property He: ndarray

Equality constraints  $He=[Ae, be]$  for the constrained zonotope.

#### Returns

H-Rep in  $[Ae, be]$ . He is `np.empty((0, self.latent_dim + 1))` for a zonotope.

#### Return type

`numpy.ndarray`

### property be: ndarray

Equality constants be for the constrained zonotope.

#### Returns

Equality constants be for the constrained zonotope. be is `np.empty((0,))` for a zonotope.

#### Return type

`numpy.ndarray`

### property c: ndarray | None

Affine transformation vector c for the constrained zonotope.

#### Returns

Affine transformation vector c.

#### Return type

`numpy.ndarray`

### property cvxpy\_args\_lp: dict[str, Any]

CVXPY arguments in use when solving a linear program

#### Returns

CVXPY arguments in use when solving a linear program. Defaults to dictionary in `py-cvxset.common.DEFAULT_CVXPY_ARGS_LP`.

#### Return type

`dict`

**property** `cvxpy_args_socp: dict[str, Any]`

CVXPY arguments in use when solving a second-order cone program

**Returns**

CVXPY arguments in use when solving a second-order cone program. Defaults to dictionary in `pycvxset.common.DEFAULT_CVXPY_ARGS_SOCP`.

**Return type**

dict

**property** `dim: int`

Dimension of the constrained zonotope.

**Returns**

Dimension of the constrained zonotope.

**Return type**

int

**Notes**

We determine dimension from  $G$ , since  $c$  is set to `None` in case of empty (constrained) zonotope.

**property** `is_bounded: bool`

Check if the constrained zonotope is bounded (which is always `True`)

**property** `is_empty: bool`

Check if the constrained zonotope is empty

**Raises**

**NotImplementedError** – Unable to solve the feasibility problem using CVXPY

**Returns**

When `True`, the polytope is empty

**Return type**

bool

**Notes**

This function may trigger a cvxpy feasibility problem if emptiness is not already known.

**property** `is_full_dimensional: bool`

Check if the affine dimension of the constrained zonotope is the same as the constrained zonotope dimension

**Returns**

`True` when the affine hull containing the constrained zonotope has the dimension `self.dim`

**Return type**

bool

**Notes**

An empty polytope is full dimensional if `dim=0`, otherwise it is not full-dimensional. See Sec. 2.1.3 of [BV04] for discussion on affine dimension. A non-empty zonotope is full-dimensional if and only if  $G$  has full row rank. A non-empty constrained zonotope is full-dimensional if and only if  $[G; A]$  has full row rank.

This function may trigger a cvxpy feasibility problem if emptiness is not already known.

**property** `is_singleton: bool`

Check if the constrained zonotope is a singleton

This function does not invoke any optimization solver, and relies on rank computations.

A non-empty constrained zonotope is a singleton if and only if latent set is empty or a singleton.

**property is\_zonotope: bool**

Check if the constrained zonotope is a zonotope

**property latent\_dim: int**

Latent dimension of the constrained zonotope.

**Returns**

Latent dimension of the constrained zonotope.

**Return type**

int

**property n\_equalities: int**

Number of equality constraints used when defining the constrained zonotope.

**Returns**

Number of equality constraints used when defining the constrained zonotope

**Return type**

int

**property type\_of\_set: str**

Return the type of set

**Returns**

Type of the set

**Return type**

str

## PYCVXSET.COMMON

The following functions are few additional methods provided with pycvxset that may be useful to the user.

<code>pycvxset.common.approximate_volume_from_grid</code>	Estimate area of a two-dimensional set using a grid of given step size
<code>pycvxset.common.is_constrained_zonotope(Q)</code>	Check if the set is a constrained zonotope
<code>pycvxset.common.is_ellipsoid(Q)</code>	Check if the set is an ellipsoid
<code>pycvxset.common.is_polytope(Q)</code>	Check if the set is a polytope
<code>pycvxset.common.prune_and_round_vertices(V)</code>	Filter through the vertices to skip any point that has another point (down in the list) that is close to it in the list.
<code>pycvxset.common.spread_points_on_a_unit_sphere</code>	Spread points on a unit sphere in n-dimensional space.

### 15.1 pycvxset.common.approximate\_volume\_from\_grid

`pycvxset.common.approximate_volume_from_grid`(*cvx\_set*: `ConstrainedZonotope` | `Ellipsoid` | `Polytope`, *area\_grid\_step\_size*: `int` | `float`) → `float`

Estimate area of a two-dimensional set using a grid of given step size

#### Parameters

- **cvx\_set** (`ConstrainedZonotope` | `Ellipsoid` | `Polytope`) – Set for which area is to be computed
- **area\_grid\_step\_size** (`int` | `float`) – Scalar step size that is constant in both dimensions

#### Raises

**ValueError** – When set is not 2-dimensional

#### Returns

Approximate area of the set

#### Return type

`float`

#### Notes

This function creates a 2D grid of points with a given step size, and computes the fraction of points that lie in the set. Consequently, the computed area is an approximation of the actual area of the set. The area of the bounding box is computed using the `minimum_volume_circumscribing_rectangle()` method associated with the set.

## 15.2 pycvxset.common.is\_constrained\_zonotope

`pycvxset.common.is_constrained_zonotope(Q: Any) → bool`

Check if the set is a constrained zonotope

### Parameters

**Q** (*object*) – Set to check

### Returns

Returns True if the set is a constrained zonotope, False otherwise

### Return type

bool

## 15.3 pycvxset.common.is\_ellipsoid

`pycvxset.common.is_ellipsoid(Q: Any) → bool`

Check if the set is an ellipsoid

### Parameters

**Q** (*object*) – Set to check

### Returns

Returns True if the set is an ellipsoid, False otherwise

### Return type

bool

## 15.4 pycvxset.common.is\_polytope

`pycvxset.common.is_polytope(Q: Any) → bool`

Check if the set is a polytope

### Parameters

**Q** (*object*) – Set to check

### Returns

Returns True if the set is a polytope, False otherwise

### Return type

bool

## 15.5 pycvxset.common.prune\_and\_round\_vertices

`pycvxset.common.prune_and_round_vertices(V: Sequence[Sequence[float]] | ndarray,  
decimal_precision: int = 3) → ndarray`

Filter through the vertices to skip any point that has another point (down in the list) that is close to it in the list.

### Parameters

- **V** (*Sequence[Sequence[float]] | np.ndarray*) – Matrix of vertices (N times self.dim)
- **decimal\_precision** (*int, optional*) – The decimal precision for rounding the vertices. Defaults to PLOTTING\_DECIMAL\_PRECISION\_CDD from pycvxset.common.constants.

### Returns

The pruned and rounded array of vertices.

**Return type**  
numpy.ndarray

## 15.6 pycvxset.common.spread\_points\_on\_a\_unit\_sphere

```
pycvxset.common.spread_points_on_a_unit_sphere(dim: int, n_points: int | None = None,
                                              cvxpy_socp_args: dict[str, Any] | None = None,
                                              verbose: bool = False, enable_warning: bool =
                                              True, save_points_on_a_unit_sphere: bool = True)
                                              → tuple[ndarray, float, ndarray]
```

Spread points on a unit sphere in n-dimensional space.

### Parameters

- **dim** (*int*) – The dimension of the sphere.
- **n\_points** (*int*) – The number of points to be spread on the unit sphere. Defaults to None, in which case, we choose
- **= (n\_points)**
- **cvxpy\_socp\_args** (*dict*) – Additional arguments to be passed to the CVXPY solver. Defaults to None, in which case the function uses DEFAULT\_CVXPY\_ARGS\_SOCP from pycvxset.common.constants.
- **verbose** (*bool*, *optional*) – Whether to print verbose output. Defaults to False.
- **enable\_warning** (*bool*, *optional*) – Enables the UserWarning. May be turned off if expected. Defaults to True.
- **save\_points\_on\_a\_unit\_sphere** (*bool*, *optional*) – Whether to save the computed points and minimum separation to a file. Defaults to SPOAUS\_SAVE\_POINTS\_ON\_A\_UNIT\_SPHERE.

### Returns

**A tuple containing three items:**

# opt\_locations (ndarray): The spread points on the unit sphere. # minimum\_separation (float): The minimum separation between the points. # opt\_locations\_first\_quad (ndarray): The spread points in the first quadrant.

**Return type**  
tuple

### Raises

- **ValueError** – If n\_points is less than  $2 * \text{dim}$ .
- **UserWarning** – If  $n\_points - 2 * \text{dim}$  is not a multiple of  $2^{\text{dim}}$ .
- **NotImplementedError** – Unable to solve the convexified problem using CVXPY
- **NotImplementedError** – Convex-concave procedure did not converge!

### Notes

This function uses the CVXPY library to solve a convex optimization problem to spread the points on the unit sphere. The spread points are returned as opt\_locations, the separation between the points is returned as separation, and the spread points in the first quadrant are returned as opt\_locations\_first\_quad.

For dim in [1, 2], the points are available in closed-form. For dim  $\geq 3$ , we solve the following non-convex

optimization problem using convex-concave procedure:

$$\begin{aligned}
 &\text{maximize} && R \\
 &\text{subject to} && R \geq 0 \\
 &&& x \geq R/2 \\
 &&& \|x_i - x_j\| \geq R, && 1 \leq i < j \leq n_{\text{points}} \\
 &&& \|x_i - e_j\| \geq R, && 1 \leq i \leq n_{\text{points}}, 1 \leq j \leq n_{\text{dim}} \\
 &&& \|x_i\| \leq 1, && 1 \leq i \leq n_{\text{points}} \\
 &&& \|x_i\| \geq 0.8, && 1 \leq i \leq n_{\text{points}}
 \end{aligned}$$

The optimization problem seeks to spread points (apart from the standard axes) in the first quadrant so that their pairwise separation is maximized, while they have a norm close to 1. The second constraint is motivated to ensure that the reflections of the points about the quadrant plane is also separated by  $R$ .

When `save_points_on_a_unit_sphere` is `True`, the function saves the computed points (all), the computed points in the first quadrant, and the minimum separation to a file named `tmp/spoaus_dim_{dim}_n_points_{n_points_int}.npz`. If the file already exists, the function loads the points and minimum separation from the file instead of recomputing them. This can save time when calling `spread_points_on_a_unit_sphere` with default values.

`pycvxset.common.approximate_volume_from_grid`(*cvx\_set*: [ConstrainedZonotope](#) | [Ellipsoid](#) | [Polytope](#), *area\_grid\_step\_size*: *int* | *float*) → *float*

Estimate area of a two-dimensional set using a grid of given step size

#### Parameters

- **cvx\_set** ([ConstrainedZonotope](#) | [Ellipsoid](#) | [Polytope](#)) – Set for which area is to be computed
- **area\_grid\_step\_size** (*int* | *float*) – Scalar step size that is constant in both dimensions

#### Raises

**ValueError** – When set is not 2-dimensional

#### Returns

Approximate area of the set

#### Return type

*float*

#### Notes

This function creates a 2D grid of points with a given step size, and computes the fraction of points that lie in the set. Consequently, the computed area is an approximation of the actual area of the set. The area of the bounding box is computed using the `minimum_volume_circumscribing_rectangle()` method associated with the set.

`pycvxset.common.is_constrained_zonotope`(*Q*: *Any*) → *bool*

Check if the set is a constrained zonotope

#### Parameters

**Q** (*object*) – Set to check

#### Returns

Returns `True` if the set is a constrained zonotope, `False` otherwise

#### Return type

*bool*

`pycvxset.common.is_ellipsoid`(*Q*: *Any*) → *bool*

Check if the set is an ellipsoid

#### Parameters

**Q** (*object*) – Set to check



### Returns

Returns True if the set is an ellipsoid, False otherwise

### Return type

bool

`pycvxset.common.is_polytope(Q: Any) → bool`

Check if the set is a polytope

### Parameters

**Q** (*object*) – Set to check

### Returns

Returns True if the set is a polytope, False otherwise

### Return type

bool

`pycvxset.common.prune_and_round_vertices(V: Sequence[Sequence[float]] | ndarray,  
decimal_precision: int = 3) → ndarray`

Filter through the vertices to skip any point that has another point (down in the list) that is close to it in the list.

### Parameters

- **V** (*Sequence[Sequence[float]] | np.ndarray*) – Matrix of vertices (N times self.dim)
- **decimal\_precision** (*int, optional*) – The decimal precision for rounding the vertices. Defaults to PLOTTING\_DECIMAL\_PRECISION\_CDD from pycvxset.common.constants.

### Returns

The pruned and rounded array of vertices.

### Return type

numpy.ndarray

`pycvxset.common.spread_points_on_a_unit_sphere(dim: int, n_points: int | None = None,  
cvxpy_socp_args: dict[str, Any] | None = None,  
verbose: bool = False, enable_warning: bool = True,  
save_points_on_a_unit_sphere: bool = True)  
→ tuple[ndarray, float, ndarray]`

Spread points on a unit sphere in n-dimensional space.

### Parameters

- **dim** (*int*) – The dimension of the sphere.
- **n\_points** (*int*) – The number of points to be spread on the unit sphere. Defaults to None, in which case, we choose
- **=** (*n\_points*)
- **cvxpy\_socp\_args** (*dict*) – Additional arguments to be passed to the CVXPY solver. Defaults to None, in which case the function uses DEFAULT\_CVXPY\_ARGS\_SOCP from pycvxset.common.constants.
- **verbose** (*bool, optional*) – Whether to print verbose output. Defaults to False.
- **enable\_warning** (*bool, optional*) – Enables the UserWarning. May be turned off if expected. Defaults to True.
- **save\_points\_on\_a\_unit\_sphere** (*bool, optional*) – Whether to save the computed points and minimum separation to a file. Defaults to SPOAUS\_SAVE\_POINTS\_ON\_A\_UNIT\_SPHERE.

### Returns

**A tuple containing three items:**

# opt\_locations (ndarray): The spread points on the unit sphere. # minimum\_separation (float): The minimum separation between the points. # opt\_locations\_first\_quad (ndarray): The spread points in the first quadrant.

**Return type**

tuple

**Raises**

- **ValueError** – If n\_points is less than 2 \* dim.
- **UserWarning** – If n\_points - 2 \* dim is not a multiple of 2^dim.
- **NotImplementedError** – Unable to solve the convexified problem using CVXPY
- **NotImplementedError** – Convex-concave procedure did not converge!

**Notes**

This function uses the CVXPY library to solve a convex optimization problem to spread the points on the unit sphere. The spread points are returned as opt\_locations, the separation between the points is returned as separation, and the spread points in the first quadrant are returned as opt\_locations\_first\_quad.

For dim in [1, 2], the points are available in closed-form. For dim >= 3, we solve the following non-convex optimization problem using convex-concave procedure:

$$\begin{aligned}
 &\text{maximize} && R \\
 &\text{subject to} && R \geq 0 \\
 & && x \geq R/2 \\
 & && \|x_i - x_j\| \geq R, && 1 \leq i < j \leq n_{\text{points}} \\
 & && \|x_i - e_j\| \geq R, && 1 \leq i \leq n_{\text{points}}, 1 \leq j \leq n_{\text{dim}} \\
 & && \|x_i\| \leq 1, && 1 \leq i \leq n_{\text{points}} \\
 & && \|x_i\| \geq 0.8, && 1 \leq i \leq n_{\text{points}}
 \end{aligned}$$

The optimization problem seeks to spread points (apart from the standard axes) in the first quadrant so that their pairwise separation is maximized, while they have a norm close to 1. The second constraint is motivated to ensure that the reflections of the points about the quadrant plane is also separated by R.

When *save\_points\_on\_a\_unit\_sphere* is True, the function saves the computed points (all), the computed points in the first quadrant, and the minimum separation to a file named *tmp/spoaus\_dim\_{dim}\_n\_points\_{n\_points\_int}.npz*. If the file already exists, the function loads the points and minimum separation from the file instead of recomputing them. This can save time when calling *spread\_points\_on\_a\_unit\_sphere* with default values.

## PYCVXSET.COMMON.CONSTANTS

The following provides the default constants used in pycvxset.

```
1  # Copyright (C) 2020-2026 Mitsubishi Electric Research Laboratories (MERL)
2  #
3  # SPDX-License-Identifier: AGPL-3.0-or-later
4
5  # Code purpose: Specify the constants to be used with cvxpy when optimization,
6  #               problems as well as testing workflows
7
8  PYCVXSET_ZERO = 1e-6 # Zero threshold for numerical stability
9  PYCVXSET_ZERO_GUROBI = 1e-5 # Zero threshold for numerical stability when using,
10 #                             GUROBI
11 # Zero threshold for vertex clustering when computing polytopic inner-approximation,
12 # of constrained zonotopes
13 PYCVXSET_ZERO_CDD = 1e-5
14 PLOTTING_DECIMAL_PRECISION_CDD = 3
15
16 # Solvers used by default
17 DEFAULT_LP_SOLVER_STR = "CLARABEL" # CLARABEL, MOSEK, CVXOPT, SCS, ECOS, GUROBI, OSQP
18 DEFAULT_SOCP_SOLVER_STR = "CLARABEL" # CLARABEL, MOSEK, CVXOPT, SCS, ECOS, GUROBI
19 DEFAULT_SDP_SOLVER_STR = "SCS" # CLARABEL, MOSEK, CVXOPT, SCS
20
21 # CVXPY args used by default (use "reoptimize": True when using GUROBI)
22 DEFAULT_CVXPY_ARGS_LP = {"solver": DEFAULT_LP_SOLVER_STR}
23 DEFAULT_CVXPY_ARGS_SOCP = {"solver": DEFAULT_SOCP_SOLVER_STR}
24 DEFAULT_CVXPY_ARGS_SDP = {"solver": DEFAULT_SDP_SOLVER_STR}
25
26 # Constants for spread_points_on_a_unit_sphere (SPOAUS)
27 SPOAUS_SLACK_TOLERANCE = 1e-8
28 SPOAUS_COST_TOLERANCE = 1e-5
29 SPOAUS_INITIAL_TAU = 1.0
30 SPOAUS_SCALING_TAU = 1.1
31 SPOAUS_TAU_MAX = 1e4
32 SPOAUS_ITERATIONS_AT_TAU_MAX = 20
33 SPOAUS_MINIMUM_NORM_VALUE_SQR = 0.8**2
34 # For SPOAUS_DIRECTIONS_PER_QUADRANT=20, we have 2D = 84, 3D = 166, 4D = 328, 5D = 650
35 SPOAUS_DIRECTIONS_PER_QUADRANT = 20
36 SPOAUS_SAVE_POINTS_ON_A_UNIT_SPHERE = True
37
38 # Time limit
39 TIME_LIMIT_FOR_CONSTRAINED_ZONOTOPE_CONTAINMENT_GUROBI_IN_S = 60
40
41 # Testing workflow constants | You could also do "GUROBI" in cvxpy.installed_solvers()
42 # "no", "minimal", "full" ("minimal", "full" need GUROBI license set up / connection,
43 # to a gurobi license server)
```

(continues on next page)

(continued from previous page)

```
40 # "full" also tests spread_points_on_a_unit_sphere with SPOAUS_DIRECTIONS_PER_
    ↳ QUADRANT for 3D
41 TESTING_STATEMENTS_INVOLVING_GUROBI = "minimal" # (Default: minimal)
42 TESTING_SHOW_PLOTS = False
43
44 # Plotting constants for polytopes
45 DEFAULT_PATCH_ARGS_2D = {"edgecolor": "k", "facecolor": "skyblue"}
46 DEFAULT_PATCH_ARGS_3D = {"edgecolor": "k", "facecolor": None}
47 DEFAULT_VERTEX_ARGS = {"visible": False, "s": 30, "marker": "o", "color": "k"}
48
49 # Default constants for qhull options
50 # Be resilient to slight degeneracies by default
51 DEFAULT_QHULL_OPTIONS = ""
52 # Prefer QHULL over CDD for vertex-facet enumerations (plot still uses CDD)
53 PREFER_QHULL_OVER_CDD = False
```

## TUTORIALS

We provide several tutorials to help you get started with *pycvxset*. The tutorials are designed to be run in a Jupyter notebook, which allows you to interact with the code and experiment with the library. The tutorials are available in the *docs/examples/* directory of *pycvxset*.

For your convenience, we provide a link to the tutorials below:

1. Polytope
2. ConstrainedZonotope
3. Application: Reachability using Polytope/ConstrainedZonotope classes
4. Code snippets used in ACC 2025 tool paper

In the above tutorials, we demonstrate how to use *pycvxset* to perform various operations on polytopes, ellipsoids, and constrained zonotopes. We also provide an application example that demonstrates the use of *pycvxset* in reachability analysis. The tutorials are designed to be self-contained, and we encourage you to run them in a Jupyter notebook to understand the code and experiment with the library.

---

CHAPTER  
**EIGHTEEN**

---

**REFERENCES**

## BIBLIOGRAPHY

- [BV04] Boyd, Stephen, and Lieven Vandenberghe. Convex optimization. Cambridge University Press, 2004.
- [KG98] Kolmanovsky, Ilya, and Elmer G. Gilbert. “Theory and computation of disturbance invariant sets for discrete-time linear systems.” *Mathematical problems in engineering* 4, no. 4 (1998): 317-367.
- [RK22] Raghuraman, Vignesh, and Koeln, Justin P.. “Set operations and order reductions for constrained zonotopes.” *Automatica* 139 (2022): 110204.
- [SDGR16] Scott, Joseph K., Davide M. Raimondo, Giuseppe Roberto Marseglia, and Richard D. Braatz. “Constrained zonotopes: A new tool for set-based estimation and fault detection.” *Automatica* 69 (2016): 126-136.
- [VWD24] Vinod, Abraham P., Avishai Weiss, and Stefano Di Cairano. “Projection-free computation of robust controllable sets with constrained zonotopes.” *arXiv preprint arXiv:2403.13730* (2024).
- [VWS24] Vinod, Abraham P., Avishai Weiss, and Stefano Di Cairano. “Inscribing and separating an ellipsoid and a constrained zonotope: Applications in stochastic control and centering” *IEEE Conference on Decision and Control*, 2024 (accepted).
- [Matplotlib-Line3DCollection] Matplotlib documentation for Line3DCollection (used in 3D plotting) <[https://matplotlib.org/stable/api/\\_as\\_gen/mpl\\_toolkits.mplot3d.art3d.Line3DCollection.html#mpl\\_toolkits.mplot3d.art3d.Line3DCollection.set](https://matplotlib.org/stable/api/_as_gen/mpl_toolkits.mplot3d.art3d.Line3DCollection.html#mpl_toolkits.mplot3d.art3d.Line3DCollection.set)>
- [Matplotlib-Axes3D.scatter] Matplotlib documentation for Axes3D.scatter (used in 3D plotting) <[https://matplotlib.org/stable/api/\\_as\\_gen/mpl\\_toolkits.mplot3d.axes3d.Axes3D.scatter.html#mpl\\_toolkits.mplot3d.axes3d.Axes3D.scatter](https://matplotlib.org/stable/api/_as_gen/mpl_toolkits.mplot3d.axes3d.Axes3D.scatter.html#mpl_toolkits.mplot3d.axes3d.Axes3D.scatter)>
- [Matplotlib-patch] Matplotlib documentation for options to pass with Axis.add\_patch (used in 2D plotting) <[https://matplotlib.org/stable/api/\\_as\\_gen/matplotlib.patches.Patch.html#matplotlib-patches-patch](https://matplotlib.org/stable/api/_as_gen/matplotlib.patches.Patch.html#matplotlib-patches-patch)>
- [Matplotlib-scatter] Matplotlib documentation for Axis.scatter (used in 2D plotting) <[https://matplotlib.org/stable/api/\\_as\\_gen/mpl\\_toolkits.mplot3d.art3d.Line3DCollection.html#mpl\\_toolkits.mplot3d.art3d.Line3DCollection.set](https://matplotlib.org/stable/api/_as_gen/mpl_toolkits.mplot3d.art3d.Line3DCollection.html#mpl_toolkits.mplot3d.art3d.Line3DCollection.set)>
- [EllipsoidalTbx-MinVolEll] Documentation for minimum-volume-ellipsoids in Ellipsoidal Toolbox. <[https://systemanalysisdpt-cmc-msu.github.io/ellipsoids/doc/chap\\_ellcalc.html#minimum-volume-ellipsoids](https://systemanalysisdpt-cmc-msu.github.io/ellipsoids/doc/chap_ellcalc.html#minimum-volume-ellipsoids)>.
- [EllipsoidalTbx-Min\_verticesEll] Documentation for minimum-volume-ellipsoids in Ellipsoidal Toolbox. <[https://systemanalysisdpt-cmc-msu.github.io/ellipsoids/doc/chap\\_ellcalc.html#minimum-volume-ellipsoids](https://systemanalysisdpt-cmc-msu.github.io/ellipsoids/doc/chap_ellcalc.html#minimum-volume-ellipsoids)>.

## PYTHON MODULE INDEX

### p

`pycvxset.ConstrainedZonotope`, [63](#)

`pycvxset.Ellipsoid`, [44](#)

`pycvxset.Polytope`, [16](#)



## Symbols

- `__init__()` (*pycvxset.ConstrainedZonotope* method), 60, 62
- `__init__()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 64
- `__init__()` (*pycvxset.Ellipsoid* method), 42, 43
- `__init__()` (*pycvxset.Ellipsoid.Ellipsoid* method), 45
- `__init__()` (*pycvxset.Polytope* method), 12, 14
- `__init__()` (*pycvxset.Polytope.Polytope* method), 17
- `__new__()` (*pycvxset.ConstrainedZonotope* method), 62
- `__new__()` (*pycvxset.Ellipsoid* method), 43
- `__new__()` (*pycvxset.Polytope* method), 15
- A**
- A* (*pycvxset.Polytope.Polytope* property), 36
- Ae* (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* property), 80
- Ae* (*pycvxset.Polytope.Polytope* property), 36
- `affine_hull()` (*pycvxset.Ellipsoid.Ellipsoid* method), 45
- `affine_map()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 64
- `affine_map()` (*pycvxset.Ellipsoid.Ellipsoid* method), 45
- `affine_map()` (*pycvxset.Polytope.Polytope* method), 17
- `approximate_pontryagin_difference()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 65
- `approximate_volume_from_grid()` (in module *pycvxset.common*), 83, 86
- B**
- b* (*pycvxset.Polytope.Polytope* property), 37
- be* (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* property), 80
- be* (*pycvxset.Polytope.Polytope* property), 37
- C**
- c* (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* property), 80
- c* (*pycvxset.Ellipsoid.Ellipsoid* property), 57
- `cartesian_product()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 66
- `chebyshev_centering()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 66
- `chebyshev_centering()` (*pycvxset.Ellipsoid.Ellipsoid* method), 45
- `chebyshev_centering()` (*pycvxset.Polytope.Polytope* method), 18
- `closest_point()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 66
- `closest_point()` (*pycvxset.Ellipsoid.Ellipsoid* method), 45
- `closest_point()` (*pycvxset.Polytope.Polytope* method), 18
- ConstrainedZonotope* (class in *pycvxset*), 59
- ConstrainedZonotope* (class in *pycvxset.ConstrainedZonotope*), 63
- `containment_constraints()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 67
- `containment_constraints()` (*pycvxset.Ellipsoid.Ellipsoid* method), 46
- `containment_constraints()` (*pycvxset.Polytope.Polytope* method), 19
- `contains()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 67
- `contains()` (*pycvxset.Ellipsoid.Ellipsoid* method), 46
- `contains()` (*pycvxset.Polytope.Polytope* method), 19
- `copy()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 68
- `copy()` (*pycvxset.Ellipsoid.Ellipsoid* method), 46
- `copy()` (*pycvxset.Polytope.Polytope* method), 20
- `cvxpy_args_lp` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* property), 80
- `cvxpy_args_lp` (*pycvxset.Ellipsoid.Ellipsoid* property), 57
- `cvxpy_args_lp` (*pycvxset.Polytope.Polytope* property), 37
- `cvxpy_args_sdp` (*pycvxset.Ellipsoid.Ellipsoid* property), 57
- `cvxpy_args_sdp` (*pycvxset.Polytope.Polytope* property), 38
- `cvxpy_args_socp` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* property), 80

`cvxpy_args_socp` (*pycvxset.Ellipsoid.Ellipsoid* property), 57  
`cvxpy_args_socp` (*pycvxset.Polytope.Polytope* property), 38

## D

`decompose_as_affine_transform_of_polytope_without_equalities` (*pycvxset.Polytope.Polytope* method), 20  
`deflate()` (*pycvxset.Ellipsoid.Ellipsoid* class method), 47  
`deflate_rectangle()` (*pycvxset.Polytope.Polytope* class method), 20  
`determine_H_rep()` (*pycvxset.Polytope.Polytope* method), 21  
`determine_V_rep()` (*pycvxset.Polytope.Polytope* method), 21  
`dim` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* property), 81  
`dim` (*pycvxset.Ellipsoid.Ellipsoid* property), 57  
`dim` (*pycvxset.Polytope.Polytope* property), 38  
`distance()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 68  
`distance()` (*pycvxset.Ellipsoid.Ellipsoid* method), 47  
`distance()` (*pycvxset.Polytope.Polytope* method), 21

## E

`Ellipsoid` (class in *pycvxset*), 41  
`Ellipsoid` (class in *pycvxset.Ellipsoid*), 44  
`extreme()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 69  
`extreme()` (*pycvxset.Ellipsoid.Ellipsoid* method), 47  
`extreme()` (*pycvxset.Polytope.Polytope* method), 22

## G

`G` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* property), 80  
`G` (*pycvxset.Ellipsoid.Ellipsoid* property), 57

## H

`H` (*pycvxset.Polytope.Polytope* property), 36  
`He` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* property), 80  
`He` (*pycvxset.Polytope.Polytope* property), 37

## I

`in_H_rep` (*pycvxset.Polytope.Polytope* property), 38  
`in_V_rep` (*pycvxset.Polytope.Polytope* property), 38  
`inflate()` (*pycvxset.Ellipsoid.Ellipsoid* class method), 47  
`inflate_ball()` (*pycvxset.Ellipsoid.Ellipsoid* class method), 48  
`interior_point()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 69  
`interior_point()` (*pycvxset.Ellipsoid.Ellipsoid* method), 48  
`interior_point()` (*pycvxset.Polytope.Polytope* method), 22

`intersection()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 69  
`intersection()` (*pycvxset.Polytope.Polytope* method), 22  
`intersection_under_inverse_affine_map()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 70  
`intersection_under_inverse_affine_map()` (*pycvxset.Polytope.Polytope* method), 23  
`intersection_with_affine_set()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 70  
`intersection_with_affine_set()` (*pycvxset.Ellipsoid.Ellipsoid* method), 48  
`intersection_with_affine_set()` (*pycvxset.Polytope.Polytope* method), 23  
`intersection_with_halfspaces()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 71  
`intersection_with_halfspaces()` (*pycvxset.Polytope.Polytope* method), 24  
`inverse_affine_map_under_invertible_matrix()` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* method), 71  
`inverse_affine_map_under_invertible_matrix()` (*pycvxset.Ellipsoid.Ellipsoid* method), 48  
`inverse_affine_map_under_invertible_matrix()` (*pycvxset.Polytope.Polytope* method), 24  
`is_bounded` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* property), 81  
`is_bounded` (*pycvxset.Ellipsoid.Ellipsoid* property), 58  
`is_bounded` (*pycvxset.Polytope.Polytope* property), 38  
`is_constrained_zonotope()` (in module *pycvxset.common*), 84, 86  
`is_ellipsoid()` (in module *pycvxset.common*), 84, 86  
`is_empty` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* property), 81  
`is_empty` (*pycvxset.Ellipsoid.Ellipsoid* property), 58  
`is_empty` (*pycvxset.Polytope.Polytope* property), 38  
`is_full_dimensional` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* property), 81  
`is_full_dimensional` (*pycvxset.Ellipsoid.Ellipsoid* property), 58  
`is_full_dimensional` (*pycvxset.Polytope.Polytope* property), 39  
`is_polytope()` (in module *pycvxset.common*), 84, 87  
`is_singleton` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* property), 81  
`is_singleton` (*pycvxset.Ellipsoid.Ellipsoid* property), 58  
`is_singleton` (*pycvxset.Polytope.Polytope* property), 39  
`is_zonotope` (*pycvxset.ConstrainedZonotope.ConstrainedZonotope* property), 82

## L

`latent_dim` (`pycvxset.ConstrainedZonotope.ConstrainedZonotope` property), 82  
`latent_dim` (`pycvxset.Ellipsoid.Ellipsoid` property), 58

## M

`maximum_volume_inscribing_ellipsoid` (`pycvxset.ConstrainedZonotope.ConstrainedZonotope` method), 71  
`maximum_volume_inscribing_ellipsoid` (`pycvxset.Ellipsoid.Ellipsoid` method), 49  
`maximum_volume_inscribing_ellipsoid` (`pycvxset.Polytope.Polytope` method), 24  
`minimize` (`pycvxset.ConstrainedZonotope.ConstrainedZonotope` method), 72  
`minimize` (`pycvxset.Ellipsoid.Ellipsoid` method), 49  
`minimize` (`pycvxset.Polytope.Polytope` method), 25  
`minimize_H_rep` (`pycvxset.Polytope.Polytope` method), 26  
`minimize_V_rep` (`pycvxset.Polytope.Polytope` method), 26  
`minimum_volume_circumscribing_ball` (`pycvxset.Ellipsoid.Ellipsoid` method), 50  
`minimum_volume_circumscribing_ellipsoid` (`pycvxset.Ellipsoid.Ellipsoid` method), 50  
`minimum_volume_circumscribing_ellipsoid` (`pycvxset.Polytope.Polytope` method), 27  
`minimum_volume_circumscribing_rectangle` (`pycvxset.ConstrainedZonotope.ConstrainedZonotope` method), 73  
`minimum_volume_circumscribing_rectangle` (`pycvxset.Ellipsoid.Ellipsoid` method), 50  
`minimum_volume_circumscribing_rectangle` (`pycvxset.Polytope.Polytope` method), 27  
`minus` (`pycvxset.ConstrainedZonotope.ConstrainedZonotope` method), 73  
`minus` (`pycvxset.Polytope.Polytope` method), 28  
module  
    `pycvxset.ConstrainedZonotope`, 63  
    `pycvxset.Ellipsoid`, 44  
    `pycvxset.Polytope`, 16

## N

`n_equalities` (`pycvxset.ConstrainedZonotope.ConstrainedZonotope` property), 82  
`n_equalities` (`pycvxset.Polytope.Polytope` property), 39  
`n_halfspaces` (`pycvxset.Polytope.Polytope` property), 39  
`n_vertices` (`pycvxset.Polytope.Polytope` property), 40  
`normalize` (`pycvxset.Polytope.Polytope` method), 28

## P

`plot` (`pycvxset.ConstrainedZonotope.ConstrainedZonotope` method), 74  
`plot` (`pycvxset.Ellipsoid.Ellipsoid` method), 51  
`plot` (`pycvxset.Polytope.Polytope` method), 29

`plot2d` (`pycvxset.Polytope.Polytope` method), 30  
`plot3d` (`pycvxset.Polytope.Polytope` method), 30  
`plus` (`pycvxset.ConstrainedZonotope.ConstrainedZonotope` method), 75  
`plus` (`pycvxset.Ellipsoid.Ellipsoid` method), 52  
`plus` (`pycvxset.Polytope.Polytope` method), 32  
`Polytope` (class in `pycvxset`), 11  
`Polytope` (class in `pycvxset.Polytope`), 16  
`polytopic_inner_approximation` (`pycvxset.ConstrainedZonotope.ConstrainedZonotope` method), 75  
`polytopic_inner_approximation` (`pycvxset.Ellipsoid.Ellipsoid` method), 52  
`polytopic_outer_approximation` (`pycvxset.ConstrainedZonotope.ConstrainedZonotope` method), 76  
`polytopic_outer_approximation` (`pycvxset.Ellipsoid.Ellipsoid` method), 53  
`project` (`pycvxset.ConstrainedZonotope.ConstrainedZonotope` method), 77  
`project` (`pycvxset.Ellipsoid.Ellipsoid` method), 53  
`project` (`pycvxset.Polytope.Polytope` method), 32  
`projection` (`pycvxset.ConstrainedZonotope.ConstrainedZonotope` method), 77  
`projection` (`pycvxset.Ellipsoid.Ellipsoid` method), 54  
`projection` (`pycvxset.Polytope.Polytope` method), 33  
`prune_and_round_vertices` (in module `pycvxset.common`), 84, 87  
`pycvxset.ConstrainedZonotope` module, 63  
`pycvxset.Ellipsoid` module, 44  
`pycvxset.Polytope` module, 16

## Q

`Q` (`pycvxset.Ellipsoid.Ellipsoid` property), 57  
`quadratic_form_as_a_symmetric_matrix` (`pycvxset.Ellipsoid.Ellipsoid` method), 54

## R

`remove_redundancies` (`pycvxset.ConstrainedZonotope.ConstrainedZonotope` method), 78

## S

`slice` (`pycvxset.ConstrainedZonotope.ConstrainedZonotope` method), 78  
`slice` (`pycvxset.Ellipsoid.Ellipsoid` method), 55  
`slice` (`pycvxset.Polytope.Polytope` method), 34  
`slice_then_projection` (`pycvxset.ConstrainedZonotope.ConstrainedZonotope` method), 78  
`slice_then_projection` (`pycvxset.Ellipsoid.Ellipsoid` method), 55

[slice\\_then\\_projection\(\)](#) (py-cvxset.Polytope.Polytope method), [34](#)  
[spread\\_points\\_on\\_a\\_unit\\_sphere\(\)](#) (in module pycvxset.common), [85](#), [87](#)  
[support\(\)](#) (pycvxset.ConstrainedZonotope.ConstrainedZonotope method), [79](#)  
[support\(\)](#) (pycvxset.Ellipsoid.Ellipsoid method), [56](#)  
[support\(\)](#) (pycvxset.Polytope.Polytope method), [35](#)

## T

[type\\_of\\_set](#) (pycvxset.ConstrainedZonotope.ConstrainedZonotope property), [82](#)  
[type\\_of\\_set](#) (pycvxset.Ellipsoid.Ellipsoid property), [58](#)  
[type\\_of\\_set](#) (pycvxset.Polytope.Polytope property), [40](#)

## V

[V](#) (pycvxset.Polytope.Polytope property), [37](#)  
[volume\(\)](#) (pycvxset.Ellipsoid.Ellipsoid method), [56](#)  
[volume\(\)](#) (pycvxset.Polytope.Polytope method), [36](#)