LinPy Documentation

Release 1.0

MINES ParisTech

August 25, 2014

CONTENTS

LinPy is a polyhedral library for Python based on isl. Integer Set Library (isl) is a C library for manipulating sets and relations of integer points bounded by linear constraints.

LinPy is a free software, licensed under the GPLv3 license. Its source code is available here.

To have an overview of LinPy's features, you may wish to read the *Tutorial*. For a comprehensive description of its functionalities, please consult the *Module Reference*.

CHAPTER

ONE

INSTALLATION

1.1 Dependencies

LinPy requires Python version 3.4 or above to work.

LinPy's one mandatory dependency is isl version 0.12 or 0.13 (it may work with other versions of isl, but this has not been tested). isl can be downloaded here or preferably, using your favorite package manager. For Debian or Ubuntu, the command to run is:

```
sudo apt-get install libisl-dev
```

For Arch Linux, run:

```
sudo pacman -S isl
```

Apart from isl, there are two optional dependencies that will maximize the use of LinPy's functions: SymPy and matplotlib. Please consult the SymPy download page and matplotlib installation instructions to install these libraries.

1.2 Install Using pip

This is the recommanded way to install LinPy, with the command:

```
sudo pip install linpy
```

1.3 Install From Source

Alternatively, LinPy can be installed from source. First, clone the public git repository:

```
git clone https://scm.cri.mines-paristech.fr/git/linpy.git
```

and build and install as usual with:

```
sudo python3 setup.py install
```

CHAPTER

TWO

TUTORIAL

This section a short introduction to some of LinPy's features. For a comprehensive description of its functionalities, please consult the *Module Reference*.

2.1 Z-Polyhedra

The following example shows how we can manipulate polyhedra using LinPy. Let us define two square polyhedra, corresponding to the sets `square1 = {(x, y) | 0 <= x <= 2, 0 <= y <= 2}` and `square2 = {(x, y) | 2 <= x <= 4, 2 <= y <= 4}`. First, we need define the symbols used, for instance with the `symbols()` function.

```
>>> from linpy import *
>>> x, y = symbols('x y')
```

Then, we can build the `Polyhedron` object `square1` from its constraints:

```
>>> square1 = Le(0, x, 2) & Le(0, y, 2)
>>> square1
And(0 <= x, x <= 2, 0 <= y, y <= 2)
```

LinPy provides comparison functions `Lt()`, `Le()`, `Eq()`, `Ne()`, `Ge()` and `Gt()` to build constraints, and logical operators `And()`, `Or()`, `Not()` to combine them. Alternatively, a polyhedron can be built from a string:

```
>>> square2 = Polyhedron('1 <= x <= 3, 1 <= y <= 3')
>>> square2
And(1 <= x, x <= 3, 1 <= y, y <= 3)
```

The usual polyhedral operations are available, including intersection:

```
>>> inter = square1.intersection(square2) # or square1 & square2
>>> inter
And(1 <= x, x <= 2, 1 <= y, y <= 2)
```

convex union:

```
>>> hull = square1.convex_union(square2)
>>> hull
And(0 <= x, 0 <= y, x <= y + 2, y <= x + 2, x <= 3, y <= 3)
```

and projection:

```
>>> proj = square1.project([y])
>>> proj
And(0 <= x, x <= 2)
```

Equality and inclusion tests are also provided. Special values `Empty` and `Universe` represent the empty and universe polyhedra.

```
>>> inter <= square1
True
>>> inter == Empty
False
```

2.2 Domains

LinPy is also able to manipulate polyhedral *domains*, that is, unions of polyhedra. An example of domain is the set union (as opposed to convex union) of polyhedra `square1` and `square2`. The result is a `Domain` object.

```
>>> union = square1.union(square2) # or square1 | square2
>>> union
Or(And(x <= 2, 0 <= x, y <= 2, 0 <= y), And(x <= 3, 1 <= x, y <= 3, 1 <= y))
>>> union <= hull
True
```

Unlike polyhedra, domains allow exact computation of union, subtraction and complementary operations.

```
>>> diff = square1.difference(square2) # or square1 - square2
>>> diff
Or(And(x == 0, 0 <= y, y <= 2), And(y == 0, 1 <= x, x <= 2))
>>> ~square1
Or(x + 1 <= 0, 3 <= x, And(0 <= x, x <= 2, y + 1 <= 0), And(0 <= x, x <= 2, 3 <= y))
```

2.3 Plotting

LinPy can use the `matplotlib` plotting library, if available, to plot bounded polyhedra and domains.

```
>>> import matplotlib.pyplot as plt
>>> from matplotlib import pylab
>>> fig = plt.figure()
>>> plot = fig.add_subplot(1, 1, 1, aspect='equal')
>>> square1.plot(plot, facecolor='red', alpha=0.3)
>>> square2.plot(plot, facecolor='blue', alpha=0.3)
>>> hull.plot(plot, facecolor='blue', alpha=0.3)
>>> pylab.show()
```

Note that you can pass a plot object to the `Domain.plot()` method, which provides great flexibility. Also, keyword arguments can be passed such as color and the degree of transparency of a polygon.

3D plots are also supported:

```
>>> import matplotlib.pyplot as plt
>>> from matplotlib import pylab
>>> from mpl_toolkits.mplot3d import Axes3D
>>> from linpy import *
>>> x, y, z = symbols('x y z')
>>> fig = plt.figure()
>>> plot = fig.add_subplot(1, 1, 1, projection='3d', aspect='equal')
>>> plot.set_title('Chamfered cube')
>>> poly = Le(0, x, 3) & Le(0, y, 3) & Le(0, z, 3) & \
        Le(z - 2, x) & Le(x, z + 2) & Le(1 - z, x) & Le(x, 5 - z) & \
```

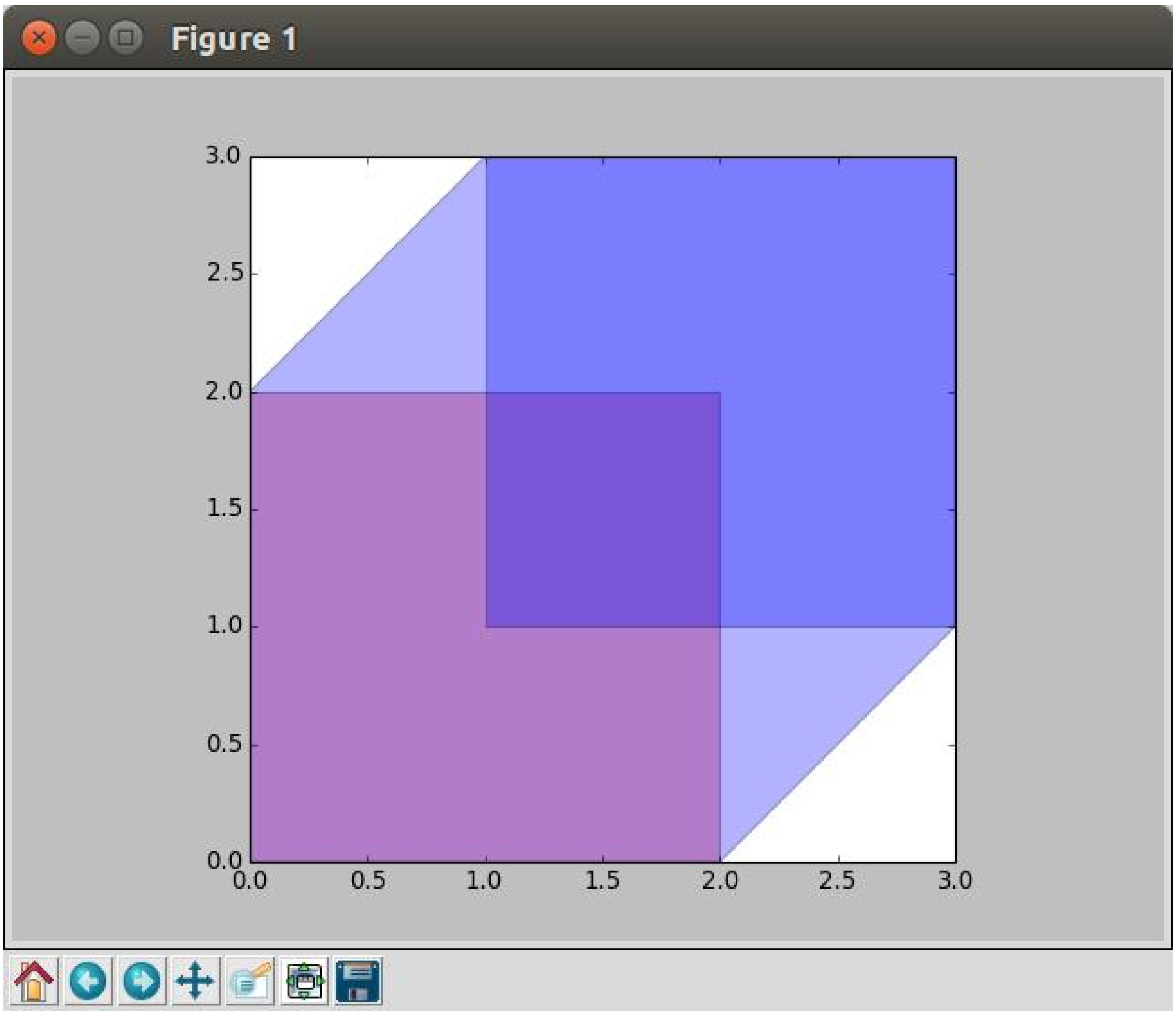
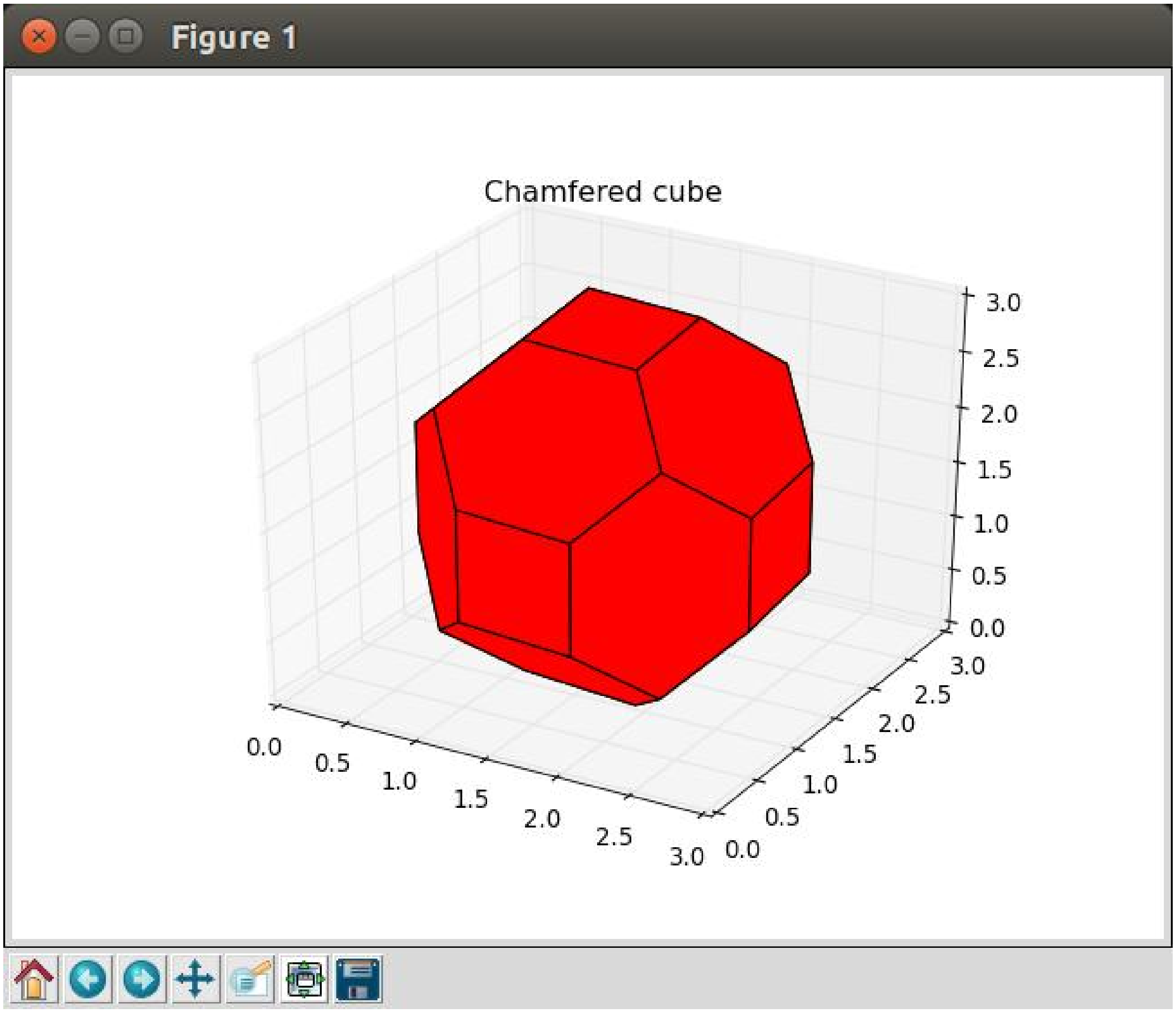


Figure 1
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

```
        Le(z - 2, y) & Le(y, z + 2) & Le(1 - z, y) & Le(y, 5 - z) & \
        Le(y - 2, x) & Le(x, y + 2) & Le(1 - y, x) & Le(x, 5 - y)
>>> poly.plot(plot, facecolor='red', alpha=0.75)
>>> pylab.show()
```



CHAPTER

THREE

MODULE REFERENCE

3.1 Symbols

Symbols are the basic components to build expressions and constraints. They correspond to mathematical variables.

class Symbol(*name*)

Return a symbol with the name string given in argument. Alternatively, the function `symbols()` allows to create several symbols at once. Symbols are instances of class `LinExpr` and inherit its functionalities.

```
>>> x = Symbol('x')
>>> x
x
```

Two instances of `Symbol` are equal if they have the same name.

name

The name of the symbol.

asdummy()

Return a new `Dummy` symbol instance with the same name.

sortkey()

Return a sorting key for the symbol. It is useful to sort a list of symbols in a consistent order, as comparison functions are overridden (see the documentation of class `LinExpr`).

```
>>> sort(symbols, key=Symbol.sortkey)
```

symbols(*names*)

This function returns a tuple of symbols whose names are taken from a comma or whitespace delimited string, or a sequence of strings. It is useful to define several symbols at once.

```
>>> x, y = symbols('x y')
>>> x, y = symbols('x, y')
>>> x, y = symbols(['x', 'y'])
```

Sometimes you need to have a unique symbol. For example, you might need a temporary one in some calculation, which is going to be substituted for something else at the end anyway. This is achieved using `Dummy('x')`.

class Dummy(*name=None*)

A variation of `Symbol` in which all symbols are unique and identified by an internal count index. If a name is not supplied then a string value of the count index will be used. This is useful when a unique, temporary variable is needed and the name of the variable used in the expression is not important.

Unlike `Symbol`, `Dummy` instances with the same name are not equal:

```
>>> x = Symbol('x')
>>> x1, x2 = Dummy('x'), Dummy('x')
>>> x == x1
False
>>> x1 == x2
False
>>> x1 == x1
True
```

3.2 Linear Expressions

A *linear expression* consists of a list of coefficient-variable pairs that capture the linear terms, plus a constant term. Linear expressions are used to build constraints. They are temporary objects that typically have short lifespans.

Linear expressions are generally built using overloaded operators. For example, if `x` is a `Symbol`, then `x + 1` is an instance of `LinExpr`.

class **`LinExpr`**(*coefficients=None*, *constant=0*)
class **`LinExpr`**(*string*)

Return a linear expression from a dictionary or a sequence, that maps symbols to their coefficients, and a constant term. The coefficients and the constant term must be rational numbers.

For example, the linear expression `x + 2*y + 1` can be constructed using one of the following instructions:

```
>>> x, y = symbols('x y')
>>> LinExpr({x: 1, y: 2}, 1)
>>> LinExpr([(x, 1), (y, 2)], 1)
```

However, it may be easier to use overloaded operators:

```
>>> x, y = symbols('x y')
>>> x + 2*y + 1
```

Alternatively, linear expressions can be constructed from a string:

```
>>> LinExpr('x + 2y + 1')
```

`LinExpr` instances are hashable, and should be treated as immutable.

A linear expression with a single symbol of coefficient 1 and no constant term is automatically subclassed as a `Symbol` instance. A linear expression with no symbol, only a constant term, is automatically subclassed as a `Rational` instance.

`coefficient`(*symbol*)
`__getitem__`(*symbol*)

Return the coefficient value of the given symbol, or `0` if the symbol does not appear in the expression.

`coefficients`()

Iterate over the pairs `(symbol, value)` of linear terms in the expression. The constant term is ignored.

`constant`

The constant term of the expression.

`symbols`

The tuple of symbols present in the expression, sorted according to `Symbol.sortkey()`.

`dimension`

The dimension of the expression, i.e. the number of symbols present in it.

isconstant()
: Return `True` if the expression only consists of a constant term. In this case, it is a `Rational` instance.

issymbol()
: Return `True` if an expression only consists of a symbol with coefficient `1`. In this case, it is a `Symbol` instance.

values()
: Iterate over the coefficient values in the expression, and the constant term.

__add__(*expr*)
: Return the sum of two linear expressions.

__sub__(*expr*)
: Return the difference between two linear expressions.

__mul__(*value*)
: Return the product of the linear expression by a rational.

__truediv__(*value*)
: Return the quotient of the linear expression by a rational.

__eq__(*expr*)
: Test whether two linear expressions are equal. Unlike methods `LinExpr.__lt__()`, `LinExpr.__le__()`, `LinExpr.__ge__()`, `LinExpr.__gt__()`, the result is a boolean value, not a polyhedron. To express that two linear expressions are equal or not equal, use functions `Eq()` and `Ne()` instead.

As explained below, it is possible to create polyhedra from linear expressions using comparison methods.

__lt__(*expr*)
__le__(*expr*)
__ge__(*expr*)
__gt__(*expr*)
: Create a new `Polyhedron` instance whose unique constraint is the comparison between two linear expressions. As an alternative, functions `Lt()`, `Le()`, `Ge()` and `Gt()` can be used.

```
>>> x, y = symbols('x y')
>>> x < y
x + 1 <= y
```

scaleint()
: Return the expression multiplied by its lowest common denominator to make all values integer.

subs(*symbol*, *expression*)
subs(*pairs*)
: Substitute the given symbol by an expression and return the resulting expression. Raise `TypeError` if the resulting expression is not linear.

```
>>> x, y = symbols('x y')
>>> e = x + 2*y + 1
>>> e.subs(y, x - 1)
3*x - 1
```

To perform multiple substitutions at once, pass a sequence or a dictionary of `(old, new)` pairs to `subs`.

```
>>> e.subs({x: y, y: x})
2*x + y + 1
```

classmethod fromstring(*string*)
: Create an expression from a string. Raise `SyntaxError` if the string is not properly formatted.

There are also methods to convert linear expressions to and from SymPy expressions:

classmethod fromsympy (*expr*)
Create a linear expression from a sympy expression. Raise TypeError is the sympy expression is not linear.

tosympy ()
Convert the linear expression to a sympy expression.

Apart from Symbol, a particular case of linear expressions are rational values, i.e. linear expressions consisting only of a constant term, with no symbol. They are implemented by the Rational class, that inherits from both LinExpr and fractions.Fraction classes.

class Rational (*numerator*, *denominator=1*)
class Rational (*string*)
The first version requires that the *numerator* and *denominator* are instances of numbers.Rational and returns a new Rational instance with the value numerator/denominator. If the denominator is 0, it raises a ZeroDivisionError. The other version of the constructor expects a string. The usual form for this instance is:

```
[sign] numerator ['/' denominator]
```

where the optional sign may be either '+' or '-' and the numerator and denominator (if present) are strings of decimal digits.

See the documentation of fractions.Fraction for more information and examples.

3.3 Polyhedra

A *convex polyhedron* (or simply "polyhedron") is the space defined by a system of linear equalities and inequalities. This space can be unbounded. A *Z-polyhedron* (simply called "polyhedron" in LinPy) is the set of integer points in a convex polyhedron.

class Polyhedron (*equalities*, *inequalities*)
class Polyhedron (*string*)
class Polyhedron (*geometric object*)
Return a polyhedron from two sequences of linear expressions: *equalities* is a list of expressions equal to 0, and *inequalities* is a list of expressions greater or equal to 0. For example, the polyhedron 0 <= x <= 2, 0 <= y <= 2 can be constructed with:

```
>>> x, y = symbols('x y')
>>> square1 = Polyhedron([], [x, 2 - x, y, 2 - y])
>>> square1
And(0 <= x, x <= 2, 0 <= y, y <= 2)
```

It may be easier to use comparison operators LinExpr.__lt__(), LinExpr.__le__(), LinExpr.__ge__(), LinExpr.__gt__(), or functions Lt(), Le(), Eq(), Ge() and Gt(), using one of the following instructions:

```
>>> x, y = symbols('x y')
>>> square1 = (0 <= x) & (x <= 2) & (0 <= y) & (y <= 2)
>>> square1 = Le(0, x, 2) & Le(0, y, 2)
```

It is also possible to build a polyhedron from a string.

```
>>> square1 = Polyhedron('0 <= x <= 2, 0 <= y <= 2')
```

Finally, a polyhedron can be constructed from a `GeometricObject` instance, calling the `GeometricObject.aspolyedron()` method. This way, it is possible to compute the polyhedral hull of a `Domain` instance, i.e., the convex hull of two polyhedra:

```
>>> square1 = Polyhedron('0 <= x <= 2, 0 <= y <= 2')
>>> square2 = Polyhedron('1 <= x <= 3, 1 <= y <= 3')
>>> Polyhedron(square1 | square2)
And(0 <= x, 0 <= y, x <= y + 2, y <= x + 2, x <= 3, y <= 3)
```

A polyhedron is a `Domain` instance, and, therefore, inherits the functionalities of this class. It is also a `GeometricObject` instance.

equalities
The tuple of equalities. This is a list of `LinExpr` instances that are equal to `0` in the polyhedron.

inequalities
The tuple of inequalities. This is a list of `LinExpr` instances that are greater or equal to `0` in the polyhedron.

constraints
The tuple of constraints, i.e., equalities and inequalities. This is semantically equivalent to: `equalities + inequalities`.

convex_union (*polyhedron*[, ...])
Return the convex union of two or more polyhedra.

asinequalities ()
Express the polyhedron using inequalities, given as a list of expressions greater or equal to 0.

widen (*polyhedron*)
Compute the *standard widening* of two polyhedra, à la Halbwachs.

In its current implementation, this method is slow and should not be used on large polyhedra.

Empty
The empty polyhedron, whose set of constraints is not satisfiable.

Universe
The universe polyhedron, whose set of constraints is always satisfiable, i.e. is empty.

3.4 Domains

A *domain* is a union of polyhedra. Unlike polyhedra, domains allow exact computation of union, subtraction and complementary operations.

class **Domain** (**polyhedra*)
class **Domain** (*string*)
class **Domain** (*geometric object*)
Return a domain from a sequence of polyhedra.

```
>>> square1 = Polyhedron('0 <= x <= 2, 0 <= y <= 2')
>>> square2 = Polyhedron('1 <= x <= 3, 1 <= y <= 3')
>>> dom = Domain(square1, square2)
>>> dom
Or(And(x <= 2, 0 <= x, y <= 2, 0 <= y), And(x <= 3, 1 <= x, y <= 3, 1 <= y))
```

It is also possible to build domains from polyhedra using arithmetic operators `Domain.__or__()`, `Domain.__invert__()` or functions `Or()` and `Not()`, using one of the following instructions:

```
>>> dom = square1 | square2
>>> dom = Or(square1, square2)
```

Alternatively, a domain can be built from a string:

```
>>> dom = Domain('0 <= x <= 2, 0 <= y <= 2; 1 <= x <= 3, 1 <= y <= 3')
```

Finally, a domain can be built from a `GeometricObject` instance, calling the `GeometricObject.asdomain()` method.

A domain is also a `GeometricObject` instance. A domain with a unique polyhedron is automatically subclassed as a `Polyhedron` instance.

polyhedra
The tuple of polyhedra present in the domain.

symbols
The tuple of symbols present in the domain equations, sorted according to `Symbol.sortkey()`.

dimension
The dimension of the domain, i.e. the number of symbols present in it.

isempty()
Return `True` if the domain is empty, that is, equal to `Empty`.

__bool__()
Return `True` if the domain is non-empty.

isuniverse()
Return `True` if the domain is universal, that is, equal to `Universe`.

isbounded()
Return `True` is the domain is bounded.

__eq__(*domain*)
Return `True` if two domains are equal.

isdisjoint(*domain*)
Return `True` if two domains have a null intersection.

issubset(*domain*)
__le__(*domain*)
Report whether another domain contains the domain.

__lt__(*domain*)
Report whether another domain is contained within the domain.

complement()
__invert__()
Return the complementary domain of the domain.

make_disjoint()
Return an equivalent domain, whose polyhedra are disjoint.

coalesce()
Simplify the representation of the domain by trying to combine pairs of polyhedra into a single polyhedron, and return the resulting domain.

detect_equalities()
Simplify the representation of the domain by detecting implicit equalities, and return the resulting domain.

remove_redundancies()
Remove redundant constraints in the domain, and return the resulting domain.

project (*symbols*)
Project out the sequence of symbols given in arguments, and return the resulting domain.

sample ()
Return a sample of the domain, as an integer instance of `Point`. If the domain is empty, a `ValueError` exception is raised.

intersection (*domain*[, ...])
__and__ (*domain*)
Return the intersection of two or more domains as a new domain. As an alternative, function `And()` can be used.

union (*domain*[, ...])
__or__ (*domain*)
__add__ (*domain*)
Return the union of two or more domains as a new domain. As an alternative, function `Or()` can be used.

difference (*domain*)
__sub__ (*domain*)
Return the difference between two domains as a new domain.

lexmin ()
Return the lexicographic minimum of the elements in the domain.

lexmax ()
Return the lexicographic maximum of the elements in the domain.

vertices ()
Return the vertices of the domain, as a list of rational instances of `Point`.

points ()
Return the integer points of a bounded domain, as a list of integer instances of `Point`. If the domain is not bounded, a `ValueError` exception is raised.

__contains__ (*point*)
Return `True` if the point is contained within the domain.

faces ()
Return the list of faces of a bounded domain. Each face is represented by a list of vertices, in the form of rational instances of `Point`. If the domain is not bounded, a `ValueError` exception is raised.

plot (*plot=None*, ***options*)
Plot a 2D or 3D domain using matplotlib. Draw it to the current *plot* object if present, otherwise create a new one. *options* are keyword arguments passed to the matplotlib drawing functions, they can be used to set the drawing color for example. Raise `ValueError` is the domain is not 2D or 3D.

subs (*symbol*, *expression*)
subs (*pairs*)
Substitute the given symbol by an expression in the domain constraints. To perform multiple substitutions at once, pass a sequence or a dictionary of `(old, new)` pairs to `subs`. The syntax of this function is similar to `LinExpr.subs()`.

classmethod fromstring (*string*)
Create a domain from a string. Raise `SyntaxError` if the string is not properly formatted.

There are also methods to convert a domain to and from SymPy expressions:

classmethod fromsympy (*expr*)
Create a domain from a sympy expression.

tosympy ()
Convert the domain to a sympy expression.

3.5 Comparison and Logic Operators

The following functions create `Polyhedron` or `Domain` instances using the comparisons of two or more `LinExpr` instances:

`Lt`(*expr1*, *expr2*[, *expr3*, ...])
Create the polyhedron with constraints `expr1 < expr2 < expr3 ...`.

`Le`(*expr1*, *expr2*[, *expr3*, ...])
Create the polyhedron with constraints `expr1 <= expr2 <= expr3 ...`.

`Eq`(*expr1*, *expr2*[, *expr3*, ...])
Create the polyhedron with constraints `expr1 == expr2 == expr3 ...`.

`Ne`(*expr1*, *expr2*[, *expr3*, ...])
Create the domain such that `expr1 != expr2 != expr3 ...`. The result is a `Domain` object, not a `Polyhedron`.

`Ge`(*expr1*, *expr2*[, *expr3*, ...])
Create the polyhedron with constraints `expr1 >= expr2 >= expr3 ...`.

`Gt`(*expr1*, *expr2*[, *expr3*, ...])
Create the polyhedron with constraints `expr1 > expr2 > expr3 ...`.

The following functions combine `Polyhedron` or `Domain` instances using logic operators:

`And`(*domain1*, *domain2*[, ...])
Create the intersection domain of the domains given in arguments.

`Or`(*domain1*, *domain2*[, ...])
Create the union domain of the domains given in arguments.

`Not`(*domain*)
Create the complementary domain of the domain given in argument.

3.6 Geometric Objects

class **`GeometricObject`**
`GeometricObject` is an abstract class to represent objects with a geometric representation in space. Subclasses of `GeometricObject` are `Polyhedron`, `Domain` and `Point`. The following elements must be provided:

`symbols`
The tuple of symbols present in the object expression, sorted according to `Symbol.sortkey()`.

`dimension`
The dimension of the object, i.e. the number of symbols present in it.

`aspolyedron`()
Return a `Polyhedron` object that approximates the geometric object.

`asdomain`()
Return a `Domain` object that approximates the geometric object.

class **`Point`**(*coordinates*)
Create a point from a dictionary or a sequence that maps the symbols to their coordinates. Coordinates must be rational numbers.

For example, the point `(x: 1, y: 2)` can be constructed using one of the following instructions:

```
>>> x, y = symbols('x y')
>>> p = Point({x: 1, y: 2})
>>> p = Point([(x, 1), (y, 2)])
```

`Point` instances are hashable and should be treated as immutable.

A point is a `GeometricObject` instance.

symbols
The tuple of symbols present in the point, sorted according to `Symbol.sortkey()`.

dimension
The dimension of the point, i.e. the number of symbols present in it.

coordinate(*symbol*)
__getitem__(*symbol*)
Return the coordinate value of the given symbol. Raise `KeyError` if the symbol is not involved in the point.

coordinates()
Iterate over the pairs `(symbol, value)` of coordinates in the point.

values()
Iterate over the coordinate values in the point.

isorigin()
Return `True` if all coordinates are `0`.

__bool__()
Return `True` if not all coordinates are `0`.

__add__(*vector*)
Translate the point by a `Vector` object and return the resulting point.

__sub__(*point*)
__sub__(*vector*)
The first version substracts a point from another and returns the resulting vector. The second version translates the point by the opposite vector of *vector* and returns the resulting point.

__eq__(*point*)
Test whether two points are equal.

class **Vector**(*coordinates*)
class **Vector**(*point1*, *point2*)
The first version creates a vector from a dictionary or a sequence that maps the symbols to their coordinates, similarly to `Point()`. The second version creates a vector between two points.

`Vector` instances are hashable and should be treated as immutable.

symbols
The tuple of symbols present in the point, sorted according to `Symbol.sortkey()`.

dimension
The dimension of the point, i.e. the number of symbols present in it.

coordinate(*symbol*)
__getitem__(*symbol*)
Return the coordinate value of the given symbol. Raise `KeyError` if the symbol is not involved in the point.

coordinates()
Iterate over the pairs `(symbol, value)` of coordinates in the point.

values()
Iterate over the coordinate values in the point.

isnull()
Return `True` if all coordinates are `0`.

__bool__()
Return `True` if not all coordinates are `0`.

__add__(*point*)
__add__(*vector*)
The first version translates the point *point* to the vector and returns the resulting point. The second version adds vector *vector* to the vector and returns the resulting vector.

__sub__(*point*)
__sub__(*vector*)
The first version substracts a point from a vector and returns the resulting point. The second version returns the difference vector between two vectors.

__neg__()
Return the opposite vector.

__mul__(*value*)
Multiply the vector by a scalar value and return the resulting vector.

__truediv__(*value*)
Divide the vector by a scalar value and return the resulting vector.

__eq__(*vector*)
Test whether two vectors are equal.

angle(*vector*)
Retrieve the angle required to rotate the vector into the vector passed in argument. The result is an angle in radians, ranging between `-pi` and `pi`.

cross(*vector*)
Compute the cross product of two 3D vectors. If either one of the vectors is not three-dimensional, a `ValueError` exception is raised.

dot(*vector*)
Compute the dot product of two vectors.

norm()
Return the norm of the vector.

norm2()
Return the squared norm of the vector.

asunit()
Return the normalized vector, i.e. the vector of same direction but with norm 1.

Symbols

A

C

D

E

F

G