pycddlib Documentation

Release 2.1.7

Matthias C. M. Troffaes

Aug 11, 2023

Contents

Release 2.1.7

Date Aug 11, 2023

CHAPTER 1

Getting Started

1.1 Overview

pycddlib is a Python wrapper for Komei Fukuda's cddlib.

cddlib is an implementation of the Double Description Method of Motzkin et al. for generating all vertices (i.e. extreme points) and extreme rays of a general convex polyhedron given by a system of linear inequalities.

The program also supports the reverse operation (i.e. convex hull computation). This means that one can move back and forth between an inequality representation and a generator (i.e. vertex and ray) representation of a polyhedron with cdd. Also, it can solve a linear programming problem, i.e. a problem of maximizing and minimizing a linear function over a polyhedron.

- Download: https://pypi.org/project/pycddlib/#files
- Documentation: https://pycddlib.readthedocs.io/en/latest/
- Development: https://github.com/mcmtroffaes/pycddlib/

1.2 Installation

1.2.1 Automatic Installer

The simplest way to install pycddlib is to install it with pip:

```
pip install pycddlib
```

On Windows, this will install from a binary wheel (for Python 3.6 and up; for older versions of Python you will need to build from source, see below).

On Linux, this will install from source, and you will need GMP as well as the Python development headers. Your distribution probably has pre-built packages for it. For example, on Fedora, install it by running:

```
dnf install gmp-devel python3-devel
```

and on Ubuntu:

```
apt-get install libgmp-dev python3-dev
```

1.2.2 Building From Source

Full build instructions are in the git repository, under python-package.yml.

For Windows, you must take care to use a compiler and platform toolset that is compatible with the one that was used to compile Python. For Python 3.6 to 3.10, you can use Visual Studio 2022 with platform toolset v143.

Next, you can build MPIR using its provided project file. For instance, for Python 3.6 to 3.10, this should work:

```
msbuild mpir-x.x.x/build.vc14/lib_mpir_gc/lib_mpir_gc.vcxproj /
↪p:Configuration=Release /p:Platform=x64 /p:PlatformToolset=v143
```

When building pycddlib, to tell Python where MPIR is located on your Windows machine, you can use:

```
python setup.py build build_ext -I<mpir_include_folder> -L<mpir_lib_folder>
```

CHAPTER 2

Numerical Representations

cdd.**get_number_type_from_value**(*value*)
Determine number type from a value.

Returns 'fraction' if the value is Rational or str, otherwise 'float'.

Return type str

cdd.**get_number_type_from_sequences**(**data*)
Determine number type from sequences.

Returns 'fraction' if all elements are Rational or str, otherwise 'float'.

Return type str

class cdd.**NumberTypeable**(*number_type='float'*)
Base class for any class which admits different numerical representations.

Parameters **number_type** (str) – The number type ('float' or 'fraction').

NumberTypeable.**make_number**(*value*)
Convert value into a number.

Parameters **value** (int, float, or str) – The value to convert.

Returns The converted value.

Return type *NumberType*

```
>>> nt = cdd.NumberTypeable('float')
>>> print(repr(nt.make_number('2/3'))) # doctest: +ELLIPSIS
0.666666666...
>>> nt = cdd.NumberTypeable('fraction')
>>> print(repr(nt.make_number('2/3'))) # doctest: +ELLIPSIS
Fraction(2, 3)
```

NumberTypeable.**number_str**(*value*)
Convert value into a string.

Parameters **value** (*NumberType*) – The value.

Returns A string for the value.

Return type str

```
>>> numbers = ['4', '2/3', '1.6', '-9/6', 1.12]
>>> nt = cdd.NumberTypeable('float')
>>> for number in numbers:
...     x = nt.make_number(number)
...     print(nt.number_str(x)) # doctest: +ELLIPSIS
4.0
0.666666666...
1.6
-1.5
1.12
>>> nt = cdd.NumberTypeable('fraction')
>>> for number in numbers:
...     x = nt.make_number(number)
...     print(nt.number_str(x))
4
2/3
8/5
-3/2
1261007895663739/1125899906842624
```

NumberTypeable.**number_repr**(*value*)

Return representation string for value.

Parameters **value** (*NumberType*) – The value.

Returns A string for the value.

Return type str

```
>>> numbers = ['4', '2/3', '1.6', '-9/6', 1.12]
>>> nt = cdd.NumberTypeable('float')
>>> for number in numbers:
...     x = nt.make_number(number)
...     print(nt.number_repr(x))
4.0
0.666666666...
1.6...
-1.5
1.12...
>>> nt = cdd.NumberTypeable('fraction')
>>> for number in numbers:
...     x = nt.make_number(number)
...     print(nt.number_repr(x))
4
'2/3'
'8/5'
'-3/2'
'1261007895663739/1125899906842624'
```

NumberTypeable.**number_cmp**(*num1*, *num2=None*)

Compare values. Type checking may not be performed, for speed. If *num2* is not specified, then *num1* is compared against zero.

Parameters

- **num1** (*NumberType*) – First value.

- **num2** (*NumberType*) – Second value.

```
>>> a = cdd.NumberTypeable('float')
>>> a.number_cmp(0.0, 5.0)
-1
>>> a.number_cmp(5.0, 0.0)
1
>>> a.number_cmp(5.0, 5.0)
0
>>> a.number_cmp(1e-30)
0
>>> a = cdd.NumberTypeable('fraction')
>>> a.number_cmp(0, 1)
-1
>>> a.number_cmp(1, 0)
1
>>> a.number_cmp(0, 0)
0
>>> a.number_cmp(a.make_number(1e-30))
1
```

`NumberTypeable.`**`number_type`**
The number type as string (`'float'` or `'fraction'`).

`NumberTypeable.`**`NumberType`**
The number type as class (`float` or `Fraction`).

CHAPTER 3

Constants

class cdd.**LPObjType**

Type of objective for a linear program.

NONE

MAX

MIN

class cdd.**LPSolverType**

Type of solver for a linear program.

CRISS_CROSS

DUAL_SIMPLEX

class cdd.**LPStatusType**

Status of a linear program.

UNDECIDED

OPTIMAL

INCONSISTENT

DUAL_INCONSISTENT

STRUC_INCONSISTENT

STRUC_DUAL_INCONSISTENT

UNBOUNDED

DUAL_UNBOUNDED

class cdd.**RepType**

Type of representation. Use *INEQUALITY* for H-representation and *GENERATOR* for V-representation.

UNSPECIFIED

INEQUALITY

GENERATOR

CHAPTER 4

Sets of Linear Inequalities and Generators

class cdd.**Matrix**(*rows*, *linear=False*, *number_type=None*)

A class for working with sets of linear constraints and extreme points.

A matrix $[b \quad -A]$ in the H-representation corresponds to a polyhedron described by

$$\begin{aligned} A_i x &\le b_i \qquad \forall i \in \{1, \dots, n\} \setminus L \\ A_i x &= b_i \qquad \forall i \in L \end{aligned}$$

where L is `lin_set` and A_i corresponds to the i-th row of A.

A matrix $[t \quad V]$ in the V-representation corresponds to a polyhedron described by

$$\mathrm{conv}\{V_i : t_i = 1\} + \mathrm{nonnegspan}\{V_i : t_i = 0, i \notin L\} + \mathrm{linspan}\{V_i : t_i = 0, i \in L\}$$

where L is `lin_set` and V_i corresponds to the i-th row of V. Here conv is the convex hull operator, $\mathrm{nonnegspan}$ is the non-negative span operator, and $\mathrm{linspan}$ is the linear span operator. All entries of t must be either 0 or 1.

Bases: `NumberTypeable`

Parameters

- **rows** (`list` of `list`s.) – The rows of the matrix. Each element can be an `int`, `float`, `Fraction`, or `str`.
- **linear** (`bool`) – Whether to add the rows to the `lin_set` or not.
- **number_type** (`str`) – The number type (`'float'` or `'fraction'`). If omitted, `get_number_type_from_sequences()` is used to determine the number type.

Warning: With the fraction number type, beware when using floats:

```
>>> print(cdd.Matrix([[1.12]], number_type='fraction')[0][0])
1261007895663739/1125899906842624
```

If the float represents a fraction, it is better to pass it as a string, so it gets automatically converted to its exact fraction representation:

```
>>> print(cdd.Matrix([['1.12']])[0][0])
28/25
```

Of course, for the float number type, both `1.12` and `'1.12'` will yield the same result, namely the `float` `1.12`.

4.1 Methods and Attributes

`Matrix.`**`__getitem__`**(*key*)

Return a row, or a slice of rows, of the matrix.

Parameters **key** (`int` or `slice`) – The row number, or slice of row numbers, to get.

Return type `tuple` of *NumberType*, or `tuple` of `tuple` of *NumberType*

`Matrix.`**`canonicalize`**()

Transform to canonical representation by recognizing all implicit linearities and all redundancies. These are returned as a pair of sets of row indices.

`Matrix.`**`copy`**()

Make a copy of the matrix and return that copy.

`Matrix.`**`extend`**(*rows*, *linear=False*)

Append rows to self (this corresponds to the dd_MatrixAppendTo function in cdd; to emulate the effect of dd_MatrixAppend, first call copy and then call extend on the copy).

The column size must be equal in the two input matrices. It raises a ValueError if the input rows are not appropriate.

Parameters

- **rows** (`list` of `lists`) – The rows to append.
- **linear** (`bool`) – Whether to add the rows to the *lin_set* or not.

`Matrix.`**`row_size`**

Number of rows.

`Matrix.`**`col_size`**

Number of columns.

`Matrix.`**`lin_set`**

A `frozenset` containing the rows of linearity (linear generators for the V-representation, and equalities for the H-representation).

`Matrix.`**`rep_type`**

Representation (see *RepType*).

`Matrix.`**`obj_type`**

Linear programming objective: maximize or minimize (see *LPObjType*).

`Matrix.`**`obj_func`**

A `tuple` containing the linear programming objective function.

4.2 Examples

Note that the following examples presume:

```
>>> import cdd
>>> from fractions import Fraction
```

4.2.1 Number Types

```
>>> cdd.Matrix([[1.5,2]]).number_type
'float'
>>> cdd.Matrix([['1.5',2]]).number_type
'fraction'
>>> cdd.Matrix([[Fraction(3, 2),2]]).number_type
'fraction'
>>> cdd.Matrix([['1.5','2']]).number_type
'fraction'
>>> cdd.Matrix([[Fraction(3, 2), Fraction(2, 1)]]).number_type
'fraction'
```

4.2.2 Fractions

Declaring matrices, and checking some attributes:

```
>>> mat1 = cdd.Matrix([['1','2'],['3','4']])
>>> mat1.NumberType
<class 'fractions.Fraction'>
>>> print(mat1)
begin
 2 2 rational
 1 2
 3 4
end
>>> mat1.row_size
2
>>> mat1.col_size
2
>>> print(mat1[0])
(1, 2)
>>> print(mat1[1])
(3, 4)
>>> print(mat1[2]) # doctest: +ELLIPSIS
Traceback (most recent call last):
  ...
IndexError: row index out of range
>>> mat1.extend([[5,6]]) # keeps number type!
>>> mat1.row_size
3
>>> print(mat1)
begin
 3 2 rational
 1 2
 3 4
 5 6
end
>>> print(mat1[0])
(1, 2)
>>> print(mat1[1])
```

(continues on next page)

(continued from previous page)

```
(3, 4)
>>> print(mat1[2])
(5, 6)
>>> mat1[1:3]
((3, 4), (5, 6))
>>> mat1[:-1]
((1, 2), (3, 4))
```

Canonicalizing:

```
>>> mat = cdd.Matrix([[2, 1, 2, 3], [0, 1, 2, 3], [3, 0, 1, 2], [0, -2, -4, -6]],␣
↪number_type='fraction')
>>> mat.canonicalize()  # oops... must specify rep_type!
Traceback (most recent call last):
    ...
ValueError: rep_type unspecified
>>> mat.rep_type = cdd.RepType.INEQUALITY
>>> mat.canonicalize()
(frozenset(...1, 3...), frozenset(...0...))
>>> print(mat)
H-representation
linearity 1  1
begin
 2 4 rational
 0 1 2 3
 3 0 1 2
end
```

Large number tests:

```
>>> print(cdd.Matrix([[10 ** 100]], number_type='fraction'))
begin
 1 1 rational
␣
↪100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
end
>>> print(cdd.Matrix([[Fraction(10 ** 100, 13 ** 102)]], number_type='fraction'))
begin
 1 1 rational
␣
↪100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪41900763375324935816337131752019220802435288507086505431825995779964082027261786966675027703685698
end
>>> cdd.Matrix([[
↪'10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪'']], number_type='fraction')[0][0]
Fraction(10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪ 1)
>>> cdd.Matrix([[
↪'10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪41900763375324935816337131752019220802435288507086505431825995779964082027261786966675027703685698
↪'']], number_type='fraction')[0][0]
Fraction(10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪␣
↪41900763375324935816337131752019220802435288507086505431825995779964082027261786966675027703685698
```

4.2.3 Floats

Declaring matrices, and checking some attributes:

```
>>> mat1 = cdd.Matrix([[1,2],[3,4]])
>>> mat1.NumberType
<... 'fractions.Fraction'>
>>> print(mat1) # doctest: +NORMALIZE_WHITESPACE
begin
 2 2 rational
 1 2
 3 4
end
>>> mat1.row_size
2
>>> mat1.col_size
2
>>> print(mat1[0])
(1, 2)
>>> print(mat1[1])
(3, 4)
>>> print(mat1[2]) # doctest: +ELLIPSIS
Traceback (most recent call last):
  ...
IndexError: row index out of range
>>> mat1.extend([[5,6]])
>>> mat1.row_size
3
>>> print(mat1) # doctest: +NORMALIZE_WHITESPACE
begin
 3 2 rational
 1 2
 3 4
 5 6
end
>>> print(mat1[0])
(1, 2)
>>> print(mat1[1])
(3, 4)
>>> print(mat1[2])
(5, 6)
>>> mat1[1:3]
((3, 4), (5, 6))
>>> mat1[:-1]
((1, 2), (3, 4))
```

Canonicalizing:

```
>>> mat = cdd.Matrix([[2, 1, 2, 3], [0, 1, 2, 3], [3, 0, 1, 2], [0, -2, -4, -6]])
>>> mat.canonicalize()  # oops... must specify rep_type!
Traceback (most recent call last):
    ...
ValueError: rep_type unspecified
>>> mat.rep_type = cdd.RepType.INEQUALITY
>>> mat.canonicalize()
(frozenset(...1, 3...), frozenset(...0...))
>>> print(mat) # doctest: +NORMALIZE_WHITESPACE
H-representation
```

(continues on next page)

(continued from previous page)

```
linearity 1  1
begin
 2 4 rational
 0 1 2 3
 3 0 1 2
end
```

CHAPTER 5

Solving Linear Programs

class cdd.**LinProg**(*mat*)

A class for solving linear programs.

Bases: *NumberTypeable*

Parameters **mat** (*Matrix*) – The matrix to load the linear program from.

5.1 Methods and Attributes

LinProg.**solve**(*solver=cdd.LPSolverType.DUAL_SIMPLEX*)

Solve linear program.

Parameters **solver** (int) – The method of solution (see *LPSolverType*).

LinProg.**dual_solution**

A tuple containing the dual solution.

LinProg.**obj_type**

Whether we are minimizing or maximizing (see *LPObjType*).

LinProg.**obj_value**

The optimal value of the objective function.

LinProg.**primal_solution**

A tuple containing the primal solution.

LinProg.**solver**

The type of solver to use (see *LPSolverType*).

LinProg.**status**

The status of the linear program (see *LPStatusType*).

5.2 Example

```
>>> import cdd
>>> mat = cdd.Matrix([['4/3',-2,-1],['2/3',0,-1],[0,1,0],[0,0,1]], number_type=
↪'fraction')
>>> mat.obj_type = cdd.LPObjType.MAX
>>> mat.obj_func = (0,3,4)
>>> print(mat)
begin
 4 3 rational
 4/3 -2 -1
 2/3 0 -1
 0 1 0
 0 0 1
end
maximize
 0 3 4
>>> print(mat.obj_func)
(0, 3, 4)
>>> lp = cdd.LinProg(mat)
>>> lp.solve()
>>> lp.status == cdd.LPStatusType.OPTIMAL
True
>>> print(lp.obj_value)
11/3
>>> print(" ".join("{0}".format(val) for val in lp.primal_solution))
1/3 2/3
>>> print(" ".join("{0}".format(val) for val in lp.dual_solution))
3/2 5/2
```

CHAPTER 6

Working With Polyhedron Representations

class cdd.**Polyhedron**(*mat*)

A class for converting between representations of a polyhedron.

Bases: *NumberTypeable*

Parameters **mat** (*Matrix*) – The matrix to load the polyhedron from.

6.1 Methods and Attributes

Polyhedron.**get_inequalities**()

Get all inequalities.

Returns H-representation.

Return type *Matrix*

For a polyhedron described as *P = {x | A x <= b}*, the H-representation is the matrix *[b -A]*.

Polyhedron.**get_generators**()

Get all generators.

Returns V-representation.

Return type *Matrix*

For a polyhedron described as *P = conv(v_1, …, v_n) + nonneg(r_1, …, r_s)*, the V-representation matrix 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.

Polyhedron.**get_adjacency**()

Get the adjacencies.

Returns Adjacency list.

Return type tuple

H-representation: For each vertex, list adjacent vertices. V-representation: For each face, list adjacent faces.

Polyhedron.**get_input_adjacency**()
Get the input adjacencies.

Returns Input adjacency list.

Return type tuple

H-representation: For each face, list adjacent faces. V-representation: For each vertex, list adjacent vertices.

Polyhedron.**get_incidence**()
Get the incidences.

Returns Incidence list.

Return type tuple

H-representation: For each vertex, list adjacent faces. V-representation: For each face, list adjacent vertices.

Polyhedron.**get_input_incidence**()
Get the input incidences.

Returns Input incidence list.

Return type tuple

H-representation: For each face, list adjacent vertices. V-representation: For each vertex, list adjacent faces.

Polyhedron.**rep_type**
Representation (see *RepType*).

Note: The H-representation and/or V-representation are not guaranteed to be minimal, that is, they can still contain redundancy.

6.2 Examples

This is the sampleh1.ine example that comes with cddlib.

```
>>> import cdd
>>> mat = cdd.Matrix([[2,-1,-1,0],[0,1,0,0],[0,0,1,0]], number_type='fraction')
>>> mat.rep_type = cdd.RepType.INEQUALITY
>>> poly = cdd.Polyhedron(mat)
>>> print(poly)
begin
 3 4 rational
 2 -1 -1 0
 0 1 0 0
 0 0 1 0
end
>>> ext = poly.get_generators()
>>> print(ext)
V-representation
linearity 1  4
begin
 4 4 rational
 1 0 0 0
 1 2 0 0
 1 0 2 0
 0 0 0 1
```

(continues on next page)

(continued from previous page)

```
end
>>> print(list(ext.lin_set)) # note: first row is 0, so fourth row is 3
[3]
```

The following example illustrates how to get adjacencies and incidences.

```
>>> import cdd
>>> # We start with the H-representation for a square
>>> # 0 <= 1 + x1 (face 0)
>>> # 0 <= 1 + x2 (face 1)
>>> # 0 <= 1 - x1 (face 2)
>>> # 0 <= 1 - x2 (face 3)
>>> mat = cdd.Matrix([[1, 1, 0], [1, 0, 1], [1, -1, 0], [1, 0, -1]])
>>> mat.rep_type = cdd.RepType.INEQUALITY
>>> poly = cdd.Polyhedron(mat)
>>> # The V-representation can be printed in the usual way:
>>> gen = poly.get_generators()
>>> print(gen)
V-representation
begin
 4 3 rational
 1 1 -1
 1 1 1
 1 -1 1
 1 -1 -1
end
>>> # graphical depiction of vertices and faces:
>>> #
>>> #   2---(3)---1
>>> #   |         |
>>> #   |         |
>>> #  (0)       (2)
>>> #   |         |
>>> #   |         |
>>> #   3---(1)---0
>>> #
>>> # vertex 0 is adjacent to vertices 1 and 3
>>> # vertex 1 is adjacent to vertices 0 and 2
>>> # vertex 2 is adjacent to vertices 1 and 3
>>> # vertex 3 is adjacent to vertices 0 and 2
>>> print([list(x) for x in poly.get_adjacency()])
[[1, 3], [0, 2], [1, 3], [0, 2]]
>>> # vertex 0 is the intersection of faces (1) and (2)
>>> # vertex 1 is the intersection of faces (2) and (3)
>>> # vertex 2 is the intersection of faces (0) and (3)
>>> # vertex 3 is the intersection of faces (0) and (1)
>>> print([list(x) for x in poly.get_incidence()])
[[1, 2], [2, 3], [0, 3], [0, 1]]
>>> # face (0) is adjacent to faces (1) and (3)
>>> # face (1) is adjacent to faces (0) and (2)
>>> # face (2) is adjacent to faces (1) and (3)
>>> # face (3) is adjacent to faces (0) and (2)
>>> print([list(x) for x in poly.get_input_adjacency()])
[[1, 3], [0, 2], [1, 3], [0, 2], []]
>>> # face (0) intersects with vertices 2 and 3
>>> # face (1) intersects with vertices 0 and 3
>>> # face (2) intersects with vertices 0 and 1
```

(continues on next page)

(continued from previous page)

```
>>> # face (3) intersects with vertices 1 and 2
>>> print([list(x) for x in poly.get_input_incidence()])
[[2, 3], [0, 3], [0, 1], [1, 2], []]
>>> # add a vertex, and construct new polyhedron
>>> gen.extend([[1, 0, 2]])
>>> vpoly = cdd.Polyhedron(gen)
>>> print(vpoly.get_inequalities())
H-representation
begin
 5 3 rational
 1 0 1
 2 1 -1
 1 1 0
 2 -1 -1
 1 -1 0
end
>>> # so now we have:
>>> # 0 <= 1 + x2
>>> # 0 <= 2 + x1 - x2
>>> # 0 <= 1 + x1
>>> # 0 <= 2 - x1 - x2
>>> # 0 <= 1 - x1
>>> #
>>> # graphical depiction of vertices and faces:
>>> #
>>> #         4
>>> #        / \
>>> #       /   \
>>> #     (1)   (3)
>>> #     /       \
>>> #    2         1
>>> #    |         |
>>> #    |         |
>>> #   (2)       (4)
>>> #    |         |
>>> #    |         |
>>> #    3---(0)---0
>>> #
>>> # for each face, list adjacent faces
>>> print([list(x) for x in vpoly.get_adjacency()])
[[2, 4], [2, 3], [0, 1], [1, 4], [0, 3]]
>>> # for each face, list adjacent vertices
>>> print([list(x) for x in vpoly.get_incidence()])
[[0, 3], [2, 4], [2, 3], [1, 4], [0, 1]]
>>> # for each vertex, list adjacent vertices
>>> print([list(x) for x in vpoly.get_input_adjacency()])
[[1, 3], [0, 4], [3, 4], [0, 2], [1, 2]]
>>> # for each vertex, list adjacent faces
>>> print([list(x) for x in vpoly.get_input_incidence()])
[[0, 4], [3, 4], [1, 2], [0, 2], [1, 3]]
```

CHAPTER 7

Changes

7.1 Version 2.1.7 (11 August 2023)

- Specify minimum required Cython version in setup script (see issue #55, reported by sguysc).
- Fix Cython DEF syntax warning.
- Support Python 3.11, drop Python 3.6.

7.2 Version 2.1.6 (8 May 2022)

- Bump cddlib to latest git (f83bdbcbefbef960d8fb5afc282ac7c32dcbb482).
- Switch testing from appveyor to github actions.
- Fix release tarballs for recent linux/macos (see issues #49, #53, #54).

7.3 Version 2.1.5 (30 November 2021)

- Add Python 3.10 support.

7.4 Version 2.1.4 (4 January 2020)

- Extra release to fix botched tgz upload on pypi.

7.5 Version 2.1.3 (4 January 2020)

- Update for cddlib 0.94m.

- Drop Python 3.5 support. Add Python 3.9 support.

7.6 Version 2.1.2 (11 August 2020)

- Drop Python 2.7 support.
- Fix string truncation issue (see issue #39).

7.7 Version 2.1.1 (16 January 2020)

- Expose adjacency and incidence (see issues #33, #34, and #36, contributed by bobmyhill).
- Add Python 3.8 support.
- Drop Python 3.4 support.
- Use pytest instead of nose for regression tests.

7.8 Version 2.1.0 (15 October 2018)

- updated for cddlib 0.94i
- fix Cython setup requirement (see issue #27)
- add documentation about representation types (see issues #29 and #30, contributed by stephane-caron)
- add Python 3.7 support

7.9 Version 2.0.0 (13 December 2017)

- fix creation of rational matrices from numpy array's (see issues #20 and #21, reported and fixed by Hervé Audren)
- consider all numbers.Rational subtypes as rationals (instead of just Fraction)

7.10 Version 1.0.6 (24 October 2017)

- fix segfault when setting rep_type (see issues #16 and #17, reported and fixed by Hervé Audren)
- drop Python 3.3 support
- add Python 3.6 support
- updated for MPIR 3.0.0

7.11 Version 1.0.5 (24 November 2015)

- drop Python 3.2 support
- add Python 3.4 and Python 3.5 support
- Matrix.canonicalize now requires rep_type to be specified; you can get back the old behaviour by setting rep_type to cdd.RepType.INEQUALITY before calling canonicalize (reported by Stéphane Caron, fixes issue #4).
- updated for cddlib 0.94h
- windows builds now tested on appveyor
- windows wheels provided for Python 2.7, 3.3, 3.4, and 3.5
- updated for MPIR 2.7.2

7.12 Version 1.0.4 (9 July 2012)

- updated for Cython 0.16
- updated for cddlib 0.94g
- updated for MPIR 2.5.1
- various fixes in documentation
- building the documentation no longer requires cdd to be installed
- documentation hosted on readthedocs.org
- development model uses gitflow
- build script uses virtualenv
- workaround for Microsoft tmpfile bug on Vista/Win7 (reported by Lorenzo Di Gregorio)

7.13 Version 1.0.3 (24 August 2010)

- added Matrix.canonicalize method
- sanitized NumberTypeable class: no more __cinit__ magic: derived classes can decide to call __init__ or not
- improved Matrix constructor: number type is derived from the type of the elements passed to the constructor, so in general, there is no need any more to pass a number_type argument (although this still remains supported)
- added get_number_type_from_value and get_number_type_from_sequences functions to aid subclasses to determine their number type.

7.14 Version 1.0.2 (9 August 2010)

- new NumberTypeable base class to allow different representations to be delegated to construction
- everything is now contained in the cdd module
- code refactored and better organized

7.15 Version 1.0.1 (1 August 2010)

- minor documentation updates
- also support the GMPRATIONAL build of cddlib with Python's fractions.Fraction
- using MPIR so it also builds on Windows
- removed trailing newlines in __str__ methods
- modules are now called cdd (uses float) and cddgmp (uses Fraction)

7.16 Version 1.0.0 (21 July 2010)

- first release, based on cddlib 0.94f

CHAPTER 8

License

Index

Symbols

C

D

E

G

I

L

M

N

O

P

R

S

U