pycddlib Documentation

Release 1.0.2

Matthias C. M. Troffaes

August 09, 2010

CONTENTS

Release 1.0.2

Date August 09, 2010

CHAPTER

ONE

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.

1.2 Installation

1.2.1 Automatic Installer

The simplest way to install pycddlib, is to download an installer matching your version of Python, and run it.

1.2.2 Building From Source

MPIR

To compile pycddlib, you need MPIR. On Linux, your distributions probably has a pre-built package for it. For example, on Fedora, install it by running:

```
yum install mpir-devel
```

On Windows, download the latest MPIR source tarball (decompress the `mpir-x.x.x.tar.bz2` file with 7-Zip), and follow the instructions in `mpir-x.x.x\build.vc9\readme.txt`. [1] For pycddlib, you only need to build the **lib_mpir_gc** project. Once built, go to the `build.vc9\lib\win32\release` folder, and copy `mpir.h` to:

```
C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\include
```

and `mpir.lib` and `mpir.pdb` to:

[1] When compiling extension modules, it is easiest to use same compiler that was used to compile Python. For Python 2.6, 2.7, 3.0, and 3.1, this is Microsoft Visual C/C++ 2008 (the express edition will do just fine).

```
C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\lib
```

pycddlib

Once MPIR is installed, download and extract the source .zip. On Windows, start the MSVC command line, and run the setup script from within the extracted folder:

```
cd ....\pycddlib-x.x.x
C:\PythonXX\python.exe setup.py install
```

On Linux, start a terminal and run:

```
cd ..../pycddlib-x.x.x
python setup.py build
su -c 'python setup.py install'
```

1.2.3 Building From Git

To compile the *latest* code, clone the project with Git by running:

```
git clone --recursive git://github.com/mcmtroffaes/pycddlib
```

Then simply run the build.sh script: this will build the library, install it, generate the documentation, and run all the doctests. Note that, besides MPIR, you also need Cython to compile the source, and Sphinx to generate the documentation.

CHAPTER

TWO

NUMERICAL REPRESENTATIONS

class cdd.**NumberTypeable**([*arg*], ...[, *number_type=None*], ...)

Base class for any class which admits different numerical representations. Instances of this class must **always** be constructed with either

- a *number_type* keyword argument, or
- a NumberTypeable instance as first (non-keyword) argument;

When subclassing NumberTypeable, there is no need to explicitely call NumberTypeable.__init__(self) in your constructor—the NumberTypeable constructor is always called automatically, and looks for the argument as described above.

Parameters

- **arg** (NumberTypeable) – Any NumberTypeable instance.
- **number_type** (str) – The number type ('float' or 'fraction').

```
>>> x = cdd.NumberTypeable(number_type='float')
>>> x.number_type
'float'
>>> y = cdd.NumberTypeable(number_type='fraction')
>>> y.number_type
'fraction'
>>> # hyperreals are not supported :-)
>>> cdd.NumberTypeable('hyperreal')
Traceback (most recent call last):
    ...
ValueError: ...
>>> z = cdd.NumberTypeable(x)
>>> z.number_type
'float'
```

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

```
>>> numbers = ['4', '2/3', '1.6', '-9/6', 1.12]
>>> nt = cdd.NumberTypeable(number_type='float')
>>> for number in numbers:
```

```
...     x = nt.make_number(number)
...     print(repr(x))
4.0
0.6666666666666663
1.6000000000000001
-1.5
1.1200000000000001
>>> nt = cdd.NumberTypeable(number_type='fraction')
>>> for number in numbers:
...     x = nt.make_number(number)
...     print(repr(x))
Fraction(4, 1)
Fraction(2, 3)
Fraction(8, 5)
Fraction(-3, 2)
Fraction(1261007895663739, 1125899906842624)
```

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(number_type='float')
>>> for number in numbers:
...     x = nt.make_number(number)
...     print(nt.number_str(x))
4.0
0.666666666667
1.6
-1.5
1.12
>>> nt = cdd.NumberTypeable(number_type='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(number_type='float')
>>> for number in numbers:
```

```
...     x = nt.make_number(number)
...     print(nt.number_repr(x))
4.0
0.6666666666666663
1.6000000000000001
-1.5
1.1200000000000001
>>> nt = cdd.NumberTypeable(number_type='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(number_type='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(number_type='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.

```
>>> cdd.NumberTypeable(number_type='float').number_type
'float'
>>> cdd.NumberTypeable(number_type='fraction').number_type
'fraction'
```

NumberTypeable.**NumberType**

The number type as class.

```
>>> cdd.NumberTypeable(number_type='float').NumberType
<type 'float'>
```

```
>>> cdd.NumberTypeable(number_type='fraction').NumberType
<class 'fractions.Fraction'>
```

CHAPTER THREE

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

FOUR

SETS OF LINEAR INEQUALITIES AND GENERATORS

class `cdd.`**`Matrix`**(*rows*, *linear=False*)

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

Bases: `NumberTypeable`

Parameters

- **rows** (`list` of `lists`.) – 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.

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']], number_type='fraction')[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.`**`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 (generators of linearity space for V-representation, and equations for 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 Fractions

Declaring matrices, and checking some attributes:

```
>>> mat1 = cdd.Matrix([[1,2],[3,4]], number_type='fraction')
>>> 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])
Traceback (most recent call last):
  ...
IndexError: row index out of range
>>> mat1.extend([[5,6]])
>>> 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])
(3, 4)
>>> print(mat1[2])
(5, 6)
>>> mat1[1:3]
((3, 4), (5, 6))
>>> mat1[:-1]
((1, 2), (3, 4))
```

Some regression tests:

```
>>> cdd.Matrix([[1], [1, 2]], number_type='fraction')
Traceback (most recent call last):
    ...
ValueError: rows have different lengths

>>> mat = cdd.Matrix([[1], [2]], number_type='fraction')
>>> mat.obj_func = (0, 0)
Traceback (most recent call last):
    ...
ValueError: objective function does not match matrix column size
```

Large number tests:

```
>>> print(cdd.Matrix([[10 ** 100]], number_type='fraction'))
begin
 1 1 rational
 1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
end
>>> print(cdd.Matrix([[Fraction(10 ** 100, 13 ** 102)]], number_type='fraction'))
begin
 1 1 rational
 1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
end
>>> cdd.Matrix([['10000000000000000000000000000000000000000000000000000000000000000000000000000000000
Fraction(100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
>>> cdd.Matrix([['10000000000000000000000000000000000000000000000000000000000000000000000000000000000
Fraction(100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

4.2.2 Floats

Declaring matrices, and checking some attributes:

```
>>> mat1 = cdd.Matrix([[1,2],[3,4]], number_type='float')
>>> mat1.NumberType
<type 'float'>
>>> print(mat1)
begin
 2 2 real
 1 2
 3 4
end
>>> mat1.row_size
2
>>> mat1.col_size
2
>>> print(mat1[0])
(1.0, 2.0)
>>> print(mat1[1])
(3.0, 4.0)
>>> print(mat1[2])
Traceback (most recent call last):
  ...
IndexError: row index out of range
>>> mat1.extend([[5,6]])
>>> mat1.row_size
3
>>> print(mat1)
begin
 3 2 real
 1 2
 3 4
 5 6
end
>>> print(mat1[0])
(1.0, 2.0)
>>> print(mat1[1])
(3.0, 4.0)
>>> print(mat1[2])
(5.0, 6.0)
>>> mat1[1:3]
((3.0, 4.0), (5.0, 6.0))
>>> mat1[:-1]
((1.0, 2.0), (3.0, 4.0))
```

Some regression tests:

```
>>> cdd.Matrix([[1], [1, 2]], number_type='float')
Traceback (most recent call last):
    ...
ValueError: rows have different lengths

>>> mat = cdd.Matrix([[1], [2]], number_type='float')
>>> mat.obj_func = (0, 0)
Traceback (most recent call last):
    ...
ValueError: objective function does not match matrix column size
```

Large number tests:

```
>>> print(cdd.Matrix([[10 ** 100]], number_type='float'))
begin
 1 1 real
 1.000000000E+100
end
>>> print(cdd.Matrix([[Fraction(10 ** 100, 13 ** 90)]], number_type='float'))
begin
 1 1 real
 5.5603...E-01
end
>>> cdd.Matrix([['100000000000000000000000000000000000000000000000000000000000000000000000000000000
1e+100
>>> cdd.Matrix([['100000000000000000000000000000000000000000000000000000000000000000000000000000000
0.55603...
```

CHAPTER

FIVE

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 Examples

Note that the following examples presume:

```
>>> import cdd
```

5.2.1 Fractions

This is the testlp2.c example that comes with cddlib.

```
>>> 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
```

Another example.

```
>>> mat = cdd.Matrix([[1,-1,-1,-1],[-1,1,1,1],[0,1,0,0],[0,0,1,0],[0,0,0,1]], number_type='fraction')
>>> mat.obj_type = cdd.LPObjType.MIN
>>> mat.obj_func = (0,1,2,3)
>>> lp = cdd.LinProg(mat)
>>> lp.solve()
>>> print(lp.obj_value)
1
>>> mat.obj_func = (0,-1,-2,-3)
>>> lp = cdd.LinProg(mat)
>>> lp.solve()
>>> print(lp.obj_value)
-3
>>> mat.obj_func = (0,'1.12','1.2','1.3')
>>> lp = cdd.LinProg(mat)
>>> lp.solve()
>>> print(lp.obj_value) # 28/25 is 1.12
28/25
```

5.2.2 Floats

This is the testlp2.c example that comes with cddlib.

```
>>> mat = cdd.Matrix([['4/3',-2,-1],['2/3',0,-1],[0,1,0],[0,0,1]], number_type='float')
>>> mat.obj_type = cdd.LPObjType.MAX
>>> mat.obj_func = (0,3,4)
>>> print(mat)
```

```
begin
 4 3 real
 1.333333333E+00 -2 -1
 6.666666667E-01 0 -1
 0 1 0
 0 0 1
end
maximize
 0 3 4
>>> print(mat.obj_func)
(0.0, 3.0, 4.0)
>>> lp = cdd.LinProg(mat)
>>> lp.solve()
>>> lp.status == cdd.LPStatusType.OPTIMAL
True
>>> print(lp.obj_value)
3.66666...
>>> print(" ".join("{0}".format(val) for val in lp.primal_solution))
0.33333... 0.66666...
>>> print(" ".join("{0}".format(val) for val in lp.dual_solution))
1.5 2.5
```

Another example.

```
>>> mat = cdd.Matrix([[1,-1,-1,-1],[-1,1,1,1],[0,1,0,0],[0,0,1,0],[0,0,0,1]], number_type='float')
>>> mat.obj_type = cdd.LPObjType.MIN
>>> mat.obj_func = (0,1,2,3)
>>> lp = cdd.LinProg(mat)
>>> lp.solve()
>>> print(lp.obj_value)
1.0
>>> mat.obj_func = (0,-1,-2,-3)
>>> lp = cdd.LinProg(mat)
>>> lp.solve()
>>> print(lp.obj_value)
-3.0
>>> mat.obj_func = (0,'1.12','1.2','1.3')
>>> lp = cdd.LinProg(mat)
>>> lp.solve()
>>> print(lp.obj_value) # 28/25 is 1.12
1.12
```

CHAPTER

SIX

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`

`Polyhedron.get_generators()`

Get all generators.

Returns V-representation.

Return type `Matrix`

`Polyhedron.rep_type`

Representation (see `RepType`).

6.2 Examples

Note that the following examples presume:

```
>>> import cdd
```

6.2.1 Fractions

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

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

This is the testcdd2.c example that comes with cddlib.

```
>>> mat = cdd.Matrix([[7,-3,-0],[7,0,-3],[1,1,0],[1,0,1]], number_type='fraction')
>>> mat.rep_type = cdd.RepType.INEQUALITY
>>> print(mat)
H-representation
begin
 4 3 rational
 7 -3 0
 7 0 -3
 1 1 0
 1 0 1
end
>>> print(cdd.Polyhedron(mat).get_generators())
V-representation
begin
 4 3 rational
 1 7/3 -1
 1 -1 -1
 1 -1 7/3
 1 7/3 7/3
end
>>> # add an equality and an inequality
>>> mat.extend([[7, 1, -3]], linear=True)
>>> mat.extend([[7, -3, 1]])
>>> print(mat)
H-representation
linearity 1  5
begin
 6 3 rational
 7 -3 0
 7 0 -3
 1 1 0
 1 0 1
 7 1 -3
```

```
 7 -3 1
end
>>> print(cdd.Polyhedron(mat).get_generators())
V-representation
begin
 2 3 rational
 1 -1 2
 1 0 7/3
end
```

6.2.2 Floats

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

```
>>> mat = cdd.Matrix([[2,-1,-1,0],[0,1,0,0],[0,0,1,0]], number_type='float')
>>> mat.rep_type = cdd.RepType.INEQUALITY
>>> poly = cdd.Polyhedron(mat)
>>> print(poly)
begin
 3 4 real
 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 real
 1 0 0 0
 1 2 0 0
 1 0 2 0
 0 0 0 1
end
>>> print(list(ext.lin_set)) # note: first row is 0, so fourth row is 3
[3]
```

This is the testcdd2.c example that comes with cddlib.

```
>>> mat = cdd.Matrix([[7,-3,-0],[7,0,-3],[1,1,0],[1,0,1]], number_type='float')
>>> mat.rep_type = cdd.RepType.INEQUALITY
>>> print(mat)
H-representation
begin
 4 3 real
 7 -3 0
 7 0 -3
 1 1 0
 1 0 1
end
>>> print(cdd.Polyhedron(mat).get_generators())
V-representation
begin
 4 3 real
 1 2.333333333E+00 -1
 1 -1 -1
```

```
 1 -1 2.333333333E+00
 1 2.333333333E+00 2.333333333E+00
end
>>> # add an equality and an inequality
>>> mat.extend([[7, 1, -3]], linear=True)
>>> mat.extend([[7, -3, 1]])
>>> print(mat)
H-representation
linearity 1  5
begin
 6 3 real
 7 -3 0
 7 0 -3
 1 1 0
 1 0 1
 7 1 -3
 7 -3 1
end
>>> print(cdd.Polyhedron(mat).get_generators())
V-representation
begin
 2 3 real
 1 -1 2
 1 0 2.333333333E+00
end
```

INDEX