improb Documentation

Release 0.1.0

Matthias C. M. Troffaes

August 24, 2010

CONTENTS

Release 0.1.0

Date August 24, 2010

CHAPTER

ONE

GETTING STARTED

1.1 Overview

improb is a Python module for working with imprecise probabilities.

1.2 Installation

1.2.1 Automatic Installer

The simplest way to install improb, is to download the installer and run it. To use the library, you also need pycddlib.

1.2.2 Building From Source

To install from the latest source code, clone it with Git by running:

```
git clone git://github.com/mcmtroffaes/improb
```

You can also browse the source code on GitHub: mcmtroffaes/improb.

Then simply run the `build.sh` script: this will build the library, install it, generate the documentation, and run all the doctests. Note that you need Sphinx to generate the documentation and to run the doctests.

If you merely want to install the library from source, just run:

```
python setup.py install
```

CHAPTER

TWO

INTRODUCTION

To get started quickly, use

- `float` or `str` for *values*,
- `int` or `list` for *possibility spaces*,
- `list` or `dict` for *gambles*, and
- `list` for *events*.

The library supports many more modes of operation, to which we turn next.

2.1 Values

Throughout the library, you can choose (slow) exact arithmetic with `fractions.Fraction` (rational numbers), or (fast) approximate arithmetic with `float` (limited precision reals).

Numbers can be specified directly as instances of `fractions.Fraction` (e.g. `fractions.Fraction(1, 3)`) or `float` (e.g. `1.23`). You can also use `int` (e.g. `20`), or `str` in the form of for instance `'1.23'` or `'1/3'`—these are internally converted to their exact representation if you work with fractions, or their approximate representation if you work with floats.

The constructors of lower previsions and gambles have an optional *number_type* keyword argument: if it is `'float'` then float arithmetic is used, and if it is `'fraction'` then rational arithmetic is used.

If you do not specify a *number_type* on construction, then `'float'` is used, unless all values are `fractions.Fraction` or `str`.

See Also:

`cdd.NumberTypeable` A general purpose class for objects which admit different numerical representations.

`cdd.get_number_type_from_value()` Determine the number type from values.

2.2 Possibility Spaces

Any `collections.Iterable` of immutable objects can be used to specify a possibility space. Effectively, an `collections.Iterable` *pspace* is interpreted as an ordered `set`.

For convenience, you can also construct a possibility space from any integer `n`—this is equivalent to specifying `range(n)`.

For further convenience, you can construct Cartesian products simply by specifying multiple iterables or integers.

class improb.**PSpace**(**args*)

An immutable possibility space, derived from `collections.Set` and `collections.Hashable`. This is effectively an immutable ordered set with a fancy constructor.

__init__(**args*)

Convert *args* into a possibility space.

Parameters

- **args** (`collections.Iterable` or `int`) – The components of the space.

Some examples of how components can be specified:

- A range of integers.

```
>>> list(PSpace(xrange(2, 15, 3)))
[2, 5, 8, 11, 14]
```

- A string.

```
>>> list(PSpace('abcdefg'))
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

- A list of strings.

```
>>> list(PSpace('rain cloudy sunny'.split(' ')))
['rain', 'cloudy', 'sunny']
```

- As a special case, you can also specify just a single integer. This will be converted to a tuple of integers of the corresponding length.

```
>>> list(PSpace(3))
[0, 1, 2]
```

If multiple arguments are specified, the product is calculated:

```
>>> list(PSpace(3, 'abc'))
[(0, 'a'), (0, 'b'), (0, 'c'),
 (1, 'a'), (1, 'b'), (1, 'c'),
 (2, 'a'), (2, 'b'), (2, 'c')]
>>> list(PSpace(('rain', 'cloudy', 'sunny'), ('cold', 'warm')))
[('rain', 'cold'), ('rain', 'warm'),
 ('cloudy', 'cold'), ('cloudy', 'warm'),
 ('sunny', 'cold'), ('sunny', 'warm')]
```

Duplicates are automatically removed:

```
>>> list(PSpace([2, 2, 5, 3, 9, 5, 1, 2]))
[2, 5, 3, 9, 1]
```

__repr__()

```
>>> PSpace([2, 4, 5])
PSpace([2, 4, 5])
>>> PSpace([0, 1, 2])
PSpace(3)
```

__str__()

```
>>> print(PSpace([2, 4, 5]))
2 4 5
```

classmethod make (*pspace*)

If *pspace* is a `PSpace`, then returns *pspace*. Otherwise, converts *pspace* to a `PSpace`.

Parameters

- **pspace** (`list` or similar; see *Possibility Spaces*) – The possibility space.

Returns A possibility space.

Return type `PSpace`

make_event (**args*, ***kwargs*)

If *event* is a `Event`, then checks possibility space and returns *event*. Otherwise, converts *event* to a `Event`.

If you wish to construct an event on a product space, which is itself composed of a product of events, specify its components as separate arguments—in this case, each of the components must be a sequence.

Parameters

- **event** (`list` or similar; see *Events*) – The event.
- **name** (`str`) – The name of the event (used for pretty printing).

Returns A event.

Return type `Event`

Raises `ValueError` if possibility spaces do not match

```
>>> pspace = PSpace(2, 3)
>>> print(pspace.make_event([(1, 2), (0, 1)]))
(0, 0) : 0
(0, 1) : 1
(0, 2) : 0
(1, 0) : 0
(1, 1) : 0
(1, 2) : 1
>>> print(pspace.make_event((0, 1), (2,)))
(0, 0) : 0
(0, 1) : 0
(0, 2) : 1
(1, 0) : 0
(1, 1) : 0
(1, 2) : 1
```

make_gamble (*gamble*, *number_type=None*)

If *gamble* is

- a `Gamble`, then checks possibility space and number type and returns *gamble*; if number type does not correspond, returns a copy of *gamble* with requested number type,
- an `Event`, then checks possibility space and returns the indicator of *gamble* with the correct number type,
- anything else, then construct a `Gamble` using *gamble* as data.

Parameters

- **gamble** (`dict` or similar; see *Gambles*) – The gamble.

- **number_type** (`str`) – The type to use for numbers: `'float'` or `'fraction'`. If omitted, then `get_number_type_from_sequences()` is used to determine the number type.

Returns A gamble.

Return type `Gamble`

Raises `ValueError` if possibility spaces or number types do not match

```
>>> from improb import PSpace, Event, Gamble
>>> pspace = PSpace('abc')
>>> event = Event(pspace, 'ac')
>>> gamble = event.indicator('fraction')
>>> fgamble = event.indicator() # float number type
>>> pevent = Event('ab', False)
>>> pgamble = Gamble('ab', [2, 5], number_type='fraction')
>>> print(pspace.make_gamble({'b': 1}, 'fraction'))
a : 0
b : 1
c : 0
>>> print(pspace.make_gamble(event, 'fraction'))
a : 1
b : 0
c : 1
>>> print(pspace.make_gamble(gamble, 'fraction'))
a : 1
b : 0
c : 1
>>> print(pspace.make_gamble(fgamble, 'fraction'))
a : 1
b : 0
c : 1
>>> print(pspace.make_gamble(pevent, 'fraction'))
Traceback (most recent call last):
    ...
ValueError: ...
>>> print(pspace.make_gamble(pgamble, 'fraction'))
Traceback (most recent call last):
    ...
ValueError: ...
>>> print(pspace.make_gamble({'a': 1, 'b': 0, 'c': 8}, 'fraction'))
a : 1
b : 0
c : 8
>>> print(pspace.make_gamble(range(2, 9, 3), 'fraction'))
a : 2
b : 5
c : 8
```

subsets(*event=True*, *empty=True*, *full=True*, *size=None*, *contains=False*)
Iterates over all subsets of the possibility space.

Parameters

- **event** (`list` or similar; see *Events*) – An event (optional).
- **empty** (`bool`) – Whether to include the empty event or not.
- **full** (`bool`) – Whether to include *event* or not.

- **size** (`int` or `collections.Iterable`) – Any size constraints. If specified, then *empty* and *full* are ignored.
- **contains** (`list` or similar; see *Events*) – An event that must be contained in all returned subsets.

Returns Yields all subsets.

Return type Iterator of `Event`.

```
>>> pspace = PSpace([2, 4, 5])
>>> print("\n---\n".join(str(subset) for subset in pspace.subsets()))
2 : 0
4 : 0
5 : 0
---
2 : 1
4 : 0
5 : 0
---
2 : 0
4 : 1
5 : 0
---
2 : 0
4 : 0
5 : 1
---
2 : 1
4 : 1
5 : 0
---
2 : 1
4 : 0
5 : 1
---
2 : 0
4 : 1
5 : 1
---
2 : 1
4 : 1
5 : 1
>>> print("\n---\n".join(str(subset) for subset in pspace.subsets([2, 4])))
2 : 0
4 : 0
5 : 0
---
2 : 1
4 : 0
5 : 0
---
2 : 0
4 : 1
5 : 0
---
2 : 1
4 : 1
5 : 0
>>> print("\n---\n".join(str(subset) for subset in pspace.subsets([2, 4], empty=False, 
```

```
2 : 1
4 : 0
5 : 0
---
2 : 0
4 : 1
5 : 0
>>> print("\n---\n".join(str(subset) for subset in pspace.subsets(True, contains=[4])))
2 : 0
4 : 1
5 : 0
---
2 : 1
4 : 1
5 : 0
---
2 : 0
4 : 1
5 : 1
---
2 : 1
4 : 1
5 : 1
```

2.3 Gambles

Any `collections.Mapping` or `collections.Sequence` can be used to specify a gamble. Effectively, given a possibility space *pspace*, a `collections.Mapping` *mapping* corresponds to the mapping (specified as Python dictionary):

```
{omega: mapping.get(omega, 0) for omega in pspace}
```

and a `collections.Sequence` *sequence* corresponds to:

```
{omega: value for omega, value in zip(pspace, sequence)}
```

This yields maximum flexibility so you can use the simplest possible specification for a gamble, depending on the situation.

Internally, the following class is used to represent gambles; it is an immutable `collections.Mapping`, and supports the usual pointwise arithmetic operations.

class `improb.`**`Gamble`**(*pspace*, *data*, *number_type=None*)

An immutable gamble.

```
>>> pspace = PSpace('abc')
>>> Gamble(pspace, {'a': 1, 'b': 4, 'c': 8}).number_type
'float'
>>> Gamble(pspace, [1, 2, 3]).number_type
'float'
>>> Gamble(pspace, {'a': '1/7', 'b': '4/3', 'c': '8/5'}).number_type
'fraction'
>>> Gamble(pspace, ['1', '2', '3/2']).number_type
'fraction'
>>> f1 = Gamble(pspace, {'a': 1, 'b': 4, 'c': 8}, number_type='fraction')
>>> print(f1)
a : 1
```

```
b : 4
c : 8
>>> print(f1 + 2)
a : 3
b : 6
c : 10
>>> print(f1 - 2)
a : -1
b : 2
c : 6
>>> print(f1 * 2)
a : 2
b : 8
c : 16
>>> print(f1 / 3)
a : 1/3
b : 4/3
c : 8/3
>>> f2 = Gamble(pspace, {'a': 5, 'b': 8, 'c': 7}, number_type='fraction')
>>> print(f1 + f2)
a : 6
b : 12
c : 15
>>> print(f1 - f2)
a : -4
b : -4
c : 1
>>> print(f1 * f2)
a : 5
b : 32
c : 56
>>> print(f1 / f2)
Traceback (most recent call last):
    ...
TypeError: ...
>>> event = Event(pspace, 'ac')
>>> print(f1 + event)
a : 2
b : 4
c : 9
>>> print(f1 - event)
a : 0
b : 4
c : 7
>>> print(f1 * event)
a : 1
b : 0
c : 8
>>> print(f1 / event)
Traceback (most recent call last):
    ...
TypeError: ...
```

__repr__()

```
>>> Gamble([2, 3, 4], {2: 1, 3: 4, 4: 8}, number_type='float')
Gamble(pspace=PSpace([2, 3, 4]),
```

```
              mapping={2: 1.0,
                       3: 4.0,
                       4: 8.0})
>>> Gamble([2, 3, 4], {2: '2/6', 3: '4.0', 4: 8}, number_type='fraction')
Gamble(pspace=PSpace([2, 3, 4]),
       mapping={2: '1/3',
                3: 4,
                4: 8})
```

__str__()

```
>>> pspace = PSpace('rain sun clouds'.split())
>>> print(Gamble(pspace, {'rain': -14, 'sun': 4, 'clouds': 20}, number_type='float'))
rain   : -14.0
sun    : 4.0
clouds : 20.0
```

pspace

A `PSpace` representing the possibility space.

2.4 Events

Any `collections.Iterable` can be used to specify an event. Effectively, given a possibility space *pspace*, an `collections.Iterable` corresponds to the event (specified as a Python set):

```
{omega for omega in iterable if omega in pspace}
```

For convenience, you can also specify an event as `True` (which corresponds to the full set) or `False` (which corresponds to the empty set).

Internally, the following class is used to represent events; it is an immutable `collections.Set`, and supports a few more common operations.

class improb.**Event**(*pspace*, *data=False*, *name=None*)

An immutable event.

```
>>> pspace = PSpace('abcdef')
>>> event1 = Event(pspace, 'acd')
>>> print(event1)
a : 1
b : 0
c : 1
d : 1
e : 0
f : 0
>>> event2 = Event(pspace, 'cdef')
>>> print(event2)
a : 0
b : 0
c : 1
d : 1
e : 1
f : 1
>>> print(event1 & event2)
a : 0
b : 0
```

```
c : 1
d : 1
e : 0
f : 0
>>> print(event1 | event2)
a : 1
b : 0
c : 1
d : 1
e : 1
f : 1
```

__repr__()

```
>>> pspace = PSpace([2, 3, 4])
>>> Event(pspace, [3, 4])
Event(pspace=PSpace([2, 3, 4]), elements=set([3, 4]))
```

__str__()

```
>>> pspace = PSpace('rain sun clouds'.split())
>>> print(Event(pspace, 'rain clouds'.split()))
rain   : 1
sun    : 0
clouds : 1
```

complement()

Calculate the complement of the event.

```
>>> print(Event(pspace='abcde', data='bde').complement())
a : 1
b : 0
c : 1
d : 0
e : 0
```

Returns Complement.

Return type `Event`

indicator(*number_type=None*)

Return indicator gamble for the event.

Parameters

- **number_type** (`str`) – The number type (defaults to `'float'` if omitted).

Returns Indicator gamble.

Return type `Gamble`

```
>>> pspace = PSpace(5)
>>> event = Event(pspace, [2, 4])
>>> event.indicator('fraction')
Gamble(pspace=PSpace(5),
       mapping={0: 0,
                1: 0,
                2: 1,
```

```
                    3: 0,
                    4: 1})
>>> event.indicator()
Gamble(pspace=PSpace(5),
       mapping={0: 0.0,
                1: 0.0,
                2: 1.0,
                3: 0.0,
                4: 1.0})
```

pspace

An PSpace representing the possibility space.

CHAPTER

THREE

LOWER PREVISIONS

3.1 Polyhedral Lower Previsions

class `improb.lowprev.lowpoly.`**`LowPoly`**(*pspace=None*, *mapping=None*, *lprev=None*, *uprev=None*, *prev=None*, *lprob=None*, *uprob=None*, *prob=None*, *bba=None*, *credalset=None*, *number_type=None*)

Bases: `improb.lowprev.LowPrev`

An arbitrary finitely generated lower prevision, that is, a finite intersection of half-spaces, each of which constrains the set of probability mass functions.

This class is derived from `collections.MutableMapping`: keys are (gamble, event) pairs, values are (lprev, uprev) pairs:

```
>>> lpr = LowPoly(pspace='abcd', number_type='fraction')
>>> lpr[{'a': 2, 'b': 3}, 'abcd'] = ('1.5', '1.9')
>>> lpr[{'c': 1, 'd': 8}, 'cd'] = ('1.2', None)
>>> print(lpr)
a b c d
2 3 0 0 | a b c d : [3/2  , 19/10]
0 0 1 8 |     c d : [6/5  ,      ]
```

Instead of working on the mapping directly, you can use the convenience methods `set_lower()`, `set_upper()`, and `set_precise()`:

```
>>> lpr = LowPoly(pspace='abcd', number_type='fraction')
>>> lpr.set_lower({'a': 2, 'b': 3}, '1.5')
>>> lpr.set_upper({'a': 2, 'b': 3}, '1.9')
>>> lpr.set_lower({'c': 1, 'd': 8}, '1.2', event='cd')
>>> print(lpr)
a b c d
2 3 0 0 | a b c d : [3/2  , 19/10]
0 0 1 8 |     c d : [6/5  ,      ]
```

`__init__`(*pspace=None*, *mapping=None*, *lprev=None*, *uprev=None*, *prev=None*, *lprob=None*, *uprob=None*, *prob=None*, *bba=None*, *credalset=None*, *number_type=None*)

Construct a polyhedral lower prevision on *pspace*.

Parameters

- **pspace** (`list` or similar; see *Possibility Spaces*) – The possibility space.
- **mapping** (`collections.Mapping`) – Mapping from (gamble, event) to (lower prevision, upper prevision).
- **lprev** (`collections.Mapping`) – Mapping from gamble to lower prevision.

- **uprev** (collections.Mapping) – Mapping from gamble to upper prevision.
- **prev** (collections.Mapping) – Mapping from gamble to precise prevision.
- **lprob** (collections.Mapping or collections.Sequence) – Mapping from event to lower probability.
- **uprob** (collections.Mapping or collections.Sequence) – Mapping from event to upper probability.
- **prob** (collections.Mapping or collections.Sequence) – Mapping from event to precise probability.
- **bba** (collections.Mapping) – Mapping from event to basic belief assignment (useful for constructing belief functions).
- **credalset** (collections.Sequence) – Sequence of probability mass functions.
- **number_type** (str) – The number type. If not specified, it is determined using get_number_type_from_sequences() on all values.

Generally, you can pass a dict as a keyword argument in order to initialize the lower and upper previsions and/or probabilities:

```
>>> print(LowPoly(pspace=3, mapping={
...     ((3, 1, 2), True): (1.5, None),
...     ((1, 0, -1), (1, 2)): (0.25, 0.3)}))
 0    1    2
3.0  1.0 2.0  | 0 1 2 : [1.5 ,     ]
1.0  0.0 -1.0 |   1 2 : [0.25, 0.3 ]
>>> print(LowPoly(pspace=3,
...     lprev={(1, 3, 2): 1.5, (2, 0, -1): 1},
...     uprev={(2, 0, -1): 1.9},
...     prev={(9, 8, 20): 15},
...     lprob={(1, 2): 0.2, (1,): 0.1},
...     uprob={(1, 2): 0.3, (0,): 0.9},
...     prob={(2,): '0.3'}))
 0    1    2
 0.0  0.0 1.0  | 0 1 2 : [0.3 , 0.3 ]
 0.0  1.0 0.0  | 0 1 2 : [0.1 ,     ]
 0.0  1.0 1.0  | 0 1 2 : [0.2 , 0.3 ]
 1.0  0.0 0.0  | 0 1 2 : [    , 0.9 ]
 1.0  3.0 2.0  | 0 1 2 : [1.5 ,     ]
 2.0  0.0 -1.0 | 0 1 2 : [1.0 , 1.9 ]
 9.0  8.0 20.0 | 0 1 2 : [15.0, 15.0]
```

A credal set can be specified simply as a list:

```
>>> print(LowPoly(pspace=3,
...     credalset=[['0.1', '0.45', '0.45'],
...                ['0.4', '0.3', '0.3'],
...                ['0.3', '0.2', '0.5']]))
 0     1     2
-10   10    0     | 0 1 2 : [-1,   ]
-1    -2    0     | 0 1 2 : [-1,   ]
1     1     1     | 0 1 2 : [1 , 1 ]
50/23 40/23 0     | 0 1 2 : [1 ,   ]
```

As a special case, for lower/upper/precise probabilities, if you need to set values on singletons, you can use a list instead of a dictionary:

```
>>> print(LowPoly(pspace='abc', lprob=['0.1', '0.2', '0.3']))
a b c
0 0 1 | a b c : [3/10, ]
0 1 0 | a b c : [1/5 , ]
1 0 0 | a b c : [1/10, ]
```

If the first argument is a `LowPoly` instance, then it is copied. For example:

```
>>> from improb.lowprev.lowprob import LowProb
>>> lpr = LowPoly(pspace='abc', lprob=['0.1', '0.1', '0.1'])
>>> print(lpr)
a b c
0 0 1 | a b c : [1/10,     ]
0 1 0 | a b c : [1/10,     ]
1 0 0 | a b c : [1/10,     ]
>>> lprob = LowProb(lpr)
>>> print(lprob)
a     : 1/10
  b   : 1/10
    c : 1/10
```

extend(*keys=None*, *lower=True*, *upper=True*, *algorithm='linprog'*)

Calculate coherent extension to the given keys (gamble/event pairs), using linear programming.

Parameters

- **keys** (`collections.Iterable`) – The gamble/event pairs to extend. If `None`, then extends to the full domain (for `LowPoly` this raises a `ValueError`, however some derived classes implement this if they have a finite domain).
- **lower** (`bool`) – Whether to extend the lower bounds.
- **upper** (`bool`) – Whether to extend the upper bounds.

```
>>> pspace = PSpace('xyz')
>>> lpr = LowPoly(pspace=pspace, lprob=['0.1', '0.2', '0.15'], number_type='fraction')
>>> print(lpr)
x y z
0 0 1 | x y z : [3/20,     ]
0 1 0 | x y z : [1/5 ,     ]
1 0 0 | x y z : [1/10,     ]
>>> for event in pspace.subsets(empty=False):
...     lpr.extend((subevent, event) for subevent in pspace.subsets(event))
>>> print(lpr)
x y z
0 0 0 | x y z : [0    , 0    ]
0 0 1 | x y z : [3/20 , 7/10 ]
0 1 0 | x y z : [1/5  , 3/4  ]
0 1 1 | x y z : [7/20 , 9/10 ]
1 0 0 | x y z : [1/10 , 13/20]
1 0 1 | x y z : [1/4  , 4/5  ]
1 1 0 | x y z : [3/10 , 17/20]
1 1 1 | x y z : [1    , 1    ]
0 0 0 | x y   : [0    , 0    ]
0 1 0 | x y   : [4/17 , 15/17]
1 0 0 | x y   : [2/17 , 13/17]
1 1 0 | x y   : [1    , 1    ]
0 0 0 | x   z : [0    , 0    ]
0 0 1 | x   z : [3/16 , 7/8  ]
1 0 0 | x   z : [1/8  , 13/16]
```

```
1 0 1 | x   z : [1    , 1    ]
0 0 0 | x     : [0    , 0    ]
1 0 0 | x     : [1    , 1    ]
0 0 0 |   y z : [0    , 0    ]
0 0 1 |   y z : [1/6  , 7/9  ]
0 1 0 |   y z : [2/9  , 5/6  ]
0 1 1 |   y z : [1    , 1    ]
0 0 0 |   y   : [0    , 0    ]
0 1 0 |   y   : [1    , 1    ]
0 0 0 |     z : [0    , 0    ]
0 0 1 |     z : [1    , 1    ]
```

get_coherent (*algorithm='linprog'*)
Return a coherent version, using linear programming.

get_credal_set (*event=True*)
Return extreme points of the credal set conditional on event.

Returns The extreme points.

Return type Yields a `tuple` for each extreme point.

get_lower (*gamble*, *event=True*, *algorithm='linprog'*)
Calculate lower expectation, using Algorithm 4 of Walley, Pelessoni, and Vicig (2004) [1]. The algorithm deals properly with zero probabilities.

get_matrix (*gamble=None*, *event=True*)
Matrix representing the constraints of this lower prevision conditional on the given event.

get_relevant_items (*event=True*)
Calculate the relevant items for calculating the natural extension conditional on an event.

This is part (a) (b) (c) of Algorithm 4 of Walley, Pelessoni, and Vicig (2004) [2]. Also see their Algorithm 2, which is a special case of Algorithm 4 but with event equal to the empty set.

is_avoiding_sure_loss (*algorithm='linprog'*)
Check avoiding sure loss by linear programming.

This is Algorithm 2 of Walley, Pelessoni, and Vicig (2004) [2].

set_lower (*gamble*, *lprev*, *event=True*)
Constrain the expectation of *gamble* to be at least *lprev*.

Parameters

- **gamble** (`dict` or similar; see *Gambles*) – The gamble whose expectation to bound.
- **lprev** (`float` or similar; see *Values*) – The lower bound for this expectation.

set_precise (*gamble*, *prev*, *event=True*)
Constrain the expectation of *gamble* to be exactly *prev*.

Parameters

- **gamble** (`dict` or similar; see *Gambles*) – The gamble whose expectation to bound.
- **prev** (`float` or similar; see *Values*) – The precise bound for this expectation.

set_upper (*gamble*, *uprev*, *event=True*)
Constrain the expectation of *gamble* to be at most *uprev*.

Parameters

[1] Peter Walley, Renato Pelessoni, and Paolo Vicig. Journal of Statistical Planning and Inference, 126(1):119-151, November 2004.

- **gamble** (`dict` or similar; see *Gambles*) – The gamble whose expectation to bound.
- **uprev** (`float` or similar; see *Values*) – The upper bound for this expectation.

The following examples presume:

```
>>> from improb.lowprev.lowpoly import LowPoly
>>> from improb.lowprev.lowprob import LowProb
>>> from improb.lowprev.prob import Prob
>>> from improb.decision.opt import OptLowPrevMax
```

3.1.1 Natural Extension

```
>>> lpr = LowPoly(pspace=3, number_type='fraction')
>>> print "%.6f" % lpr.get_lower([1,2,3])
1.000000
>>> print "%.6f" % lpr.get_upper([1,2,3])
3.000000
>>> lpr.set_lower([1,2,3], 1.5)
>>> lpr.set_upper([1,2,3], 2.5)
>>> print(lpr.get_matrix())
H-representation
linearity 1  1
begin
 6 4 rational
 1 -1 -1 -1
 0 1 0 0
 0 0 1 0
 0 0 0 1
 0 -1/2 1/2 3/2
 0 3/2 1/2 -1/2
end
minimize
 0 0 0 0
>>> print(lpr.get_lower([1,2,3]))
3/2
>>> print(lpr.get_upper([1,2,3]))
5/2
>>> print('\n'.join(' '.join(str(x) for x in prob) for prob in sorted(lpr.get_credal_set())))
0 1/2 1/2
0 1 0
1/4 0 3/4
1/2 1/2 0
3/4 0 1/4
```

Another example:

```
>>> lpr = LowPoly(pspace=4, number_type='fraction')
>>> lpr.set_lower([4,2,1,0], 3)
>>> lpr.set_upper([4,1,2,0], 3)
>>> lpr.is_avoiding_sure_loss()
True
>>> lpr.is_coherent()
True
>>> lpr.is_linear()
False
>>> print(lpr.get_lower([1,0,0,0]))
1/2
```

```
>>> print(lpr.get_upper([1,0,0,0]))
3/4
>>> print(lpr)
0 1 2 3
4 1 2 0 | 0 1 2 3 : [ , 3]
4 2 1 0 | 0 1 2 3 : [3,  ]
>>> opt = OptLowPrevMax(lpr)
>>> list(opt([[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]]))
[[1, 0, 0, 0], [0, 1, 0, 0]]
>>> list(opt([[0,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]]))
[[0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]
```

Another example, involving zero probabilities (see Example 9 and 11 in [2]):

```
>>> pspace = PSpace(2, 2, 2, 2, 2)
>>> a1 = pspace.make_event((0,), (0, 1), (0, 1), (0, 1), (0, 1))
>>> a2 = pspace.make_event((0, 1), (0,), (0, 1), (0, 1), (0, 1))
>>> a3 = pspace.make_event((0, 1), (0, 1), (0,), (0, 1), (0, 1))
>>> a4 = pspace.make_event((0, 1), (0, 1), (0, 1), (0,), (0, 1))
>>> a5 = pspace.make_event((0, 1), (0, 1), (0, 1), (0, 1), (0,))
>>> lpr = LowPoly(pspace, number_type='fraction')
>>> lpr[a1, True] = ('0.6', '0.6')
>>> lpr[a1.complement() | a2, True] = ('0.4', '0.4')
>>> lpr[a2 | a3, True] = ('0.8', '0.8')
>>> lpr[a3 & a4, True] = ('0.3', '0.3')
>>> lpr[a4.complement() | a5, True] = ('0.5', '0.5')
>>> lpr[a2 | a5, True] = ('0.6', '0.6')
>>> lpr.get_lower(a3)
Fraction(2, 5)
>>> lpr.get_upper(a3)
Fraction(4, 5)
>>> lpr.get_lower(a4, event=a3)
Fraction(3, 8)
>>> lpr.get_upper(a4, event=a3)
Fraction(3, 4)
>>> lpr.get_lower((a1 & a2).complement(), event=(a1 & a2) | (a1.complement() & a2.complement()))
0
```

Another more complex example:

```
>>> lpr_s = LowPoly(pspace=3, number_type='fraction')
>>> lpr_s.set_lower([1,0,0], '0.4')
>>> lpr_s.set_upper([1,0,0], '0.5')
>>> lpr_s.set_lower([0,1,0], '0.3')
>>> lpr_s.set_upper([0,1,0], '0.4')
>>> lpr_s.set_lower([0,0,1], '0.2')
>>> lpr_s.set_upper([0,0,1], '0.2')
>>> lpr_s.is_coherent()
True
>>> print(lpr_s.get_lower([1,0,0]))
2/5
>>> print(lpr_s.get_upper([1,0,0]))
1/2
>>> print(lpr_s.get_lower([0,1,0]))
3/10
>>> print(lpr_s.get_upper([0,1,0]))
2/5
>>> print(lpr_s.get_lower([0,0,1]))
1/5
```

```
>>> print(lpr_s.get_upper([0,0,1]))
1/5
>>> print(lpr_s.get_lower([-7,5,20]))
2
>>> print(lpr_s.get_upper([-7,5,20]))
16/5
>>> lpr_t_s = [LowPoly(pspace=3, number_type='fraction') for i in xrange(3)]
>>> lpr_t_s[0].set_precise([1,0,0], '0.6')
>>> lpr_t_s[0].set_precise([0,1,0], '0.3')
>>> lpr_t_s[0].set_precise([0,0,1], '0.1')
>>> lpr_t_s[0].is_coherent()
True
>>> lpr_t_s[1].set_precise([1,0,0], '0.3')
>>> lpr_t_s[1].set_precise([0,1,0], '0.4')
>>> lpr_t_s[1].set_precise([0,0,1], '0.3')
>>> lpr_t_s[1].is_coherent()
True
>>> lpr_t_s[2].set_precise([1,0,0], '0.1')
>>> lpr_t_s[2].set_precise([0,1,0], '0.4')
>>> lpr_t_s[2].set_precise([0,0,1], '0.5')
>>> lpr_t_s[2].is_coherent()
True
>>> # a gamble which is a function of s and t; s comes as first
>>> # argument (in order to match lpr_t_s): gamble_t_s[s][t]
>>> gamble_t_s = [[1,0,0], [1,0,0], [1,0,0]] # event t=0
>>> # calculate its lower prevision by marginal extension
>>> print(lpr_s.get_lower([lpr_t.get_lower(gamble_t)
...                        for lpr_t, gamble_t in zip(lpr_t_s, gamble_t_s)]))
19/50
>>> print(lpr_s.get_upper([lpr_t.get_upper(gamble_t)
...                        for lpr_t, gamble_t in zip(lpr_t_s, gamble_t_s)]))
41/100
>>> gamble_t_s = [[0,1,0], [0,1,0], [0,1,0]] # event t=1
>>> print(lpr_s.get_lower([lpr_t.get_lower(gamble_t)
...                        for lpr_t, gamble_t in zip(lpr_t_s, gamble_t_s)]))
7/20
>>> print(lpr_s.get_upper([lpr_t.get_upper(gamble_t)
...                        for lpr_t, gamble_t in zip(lpr_t_s, gamble_t_s)]))
9/25
>>> gamble_t_s = [[0,0,1], [0,0,1], [0,0,1]] # event t=2
>>> print(lpr_s.get_lower([lpr_t.get_lower(gamble_t)
...                        for lpr_t, gamble_t in zip(lpr_t_s, gamble_t_s)]))
6/25
>>> print(lpr_s.get_upper([lpr_t.get_upper(gamble_t)
...                        for lpr_t, gamble_t in zip(lpr_t_s, gamble_t_s)]))
13/50
```

3.1.2 Mobius Transform

```
>>> lpr = LowProb(pspace=2)
>>> lpr.set_lower([0,0], 0)
>>> lpr.set_lower([1,0], 0.3)
>>> lpr.set_lower([0,1], 0.2)
>>> lpr.set_lower([1,1], 1)
>>> print(lpr.mobius)
    : 0.0
0   : 0.3
```

```
  1 : 0.2
0 1 : 0.5
```

3.1.3 Frechet Bounds

```
>>> lpr = LowPoly(pspace=4, number_type='fraction')
>>> lpr.set_precise([1,1,0,0], '0.6')
>>> lpr.set_precise([0,1,1,0], '0.7')
>>> print(lpr.get_lower([0,1,0,0])) # max(0.6+0.7-1,0)
3/10
>>> print(lpr.get_upper([0,1,0,0])) # min(0.6,0.7)
3/5
>>> lpr.is_linear()
True
>>> for vert in lpr.get_credal_set():
...     print(" ".join("%.2f" % float(x) for x in vert))
0.30 0.30 0.40 0.00
0.00 0.60 0.10 0.30
```

3.1.4 Avoiding Sure Loss

This example incurs sure loss because the maximum of the sum of the gambles [1,2,3,0] and [3,2,1,0] is 4, however 2.5 + 2.5 is strictly larger than 4.

```
>>> lpr = LowPoly(pspace=4)
>>> lpr.set_lower([1,2,3,0], 2.5)
>>> lpr.set_lower([3,2,1,0], 2.5)
>>> lpr.is_avoiding_sure_loss()
False
```

A few simple examples:

```
>>> lpr = LowPoly(pspace='xyz')
>>> lpr.is_avoiding_sure_loss()
True
```

```
>>> lpr = LowPoly(pspace='xyz', lprob=['0.1', '0.2', '0.15'])
>>> lpr[{'x':1}, 'x'] = (1, None)
>>> lpr[{'x':0}, 'x'] = (0, None)
>>> lpr.is_avoiding_sure_loss()
True
```

See if we can handle zero probabilities:

```
>>> lpr = LowPoly(pspace='abcd', number_type='fraction')
>>> lpr[{'a': 1}, True] = (1, None)
>>> print(lpr)
a b c d
1 0 0 0 | a b c d : [1,  ]
>>> lpr.is_avoiding_sure_loss()
True
```

Slightly more complicated:

```
>>> lpr = LowPoly(pspace='abcd', number_type='fraction')
>>> lpr[{'a': 1}, True] = (1, None)
```

```
>>> lpr[{'b': 1}, 'bcd'] = (2, None) # obviously incurs sure loss!
>>> print(lpr)
a b c d
1 0 0 0 | a b c d : [1,  ]
0 1 0 0 |   b c d : [2,  ]
>>> lpr.is_avoiding_sure_loss()
False
```

And even slightly more complicated:

```
>>> lpr = LowPoly(pspace='abcd', number_type='fraction')
>>> lpr[{'a': 1}, True] = (1, None)
>>> lpr[{'b': 1}, 'bcd'] = ('2/3', None)
>>> lpr[{'c': 1}, 'bcd'] = ('2/3', None) # sum larger than one on this layer
>>> print(lpr)
a b c d
1 0 0 0 | a b c d : [1  ,    ]
0 0 1 0 |   b c d : [2/3,    ]
0 1 0 0 |   b c d : [2/3,    ]
>>> lpr.is_avoiding_sure_loss()
False
```

Another more complex example:

```
>>> lpr_s_t = LowPoly(pspace=9)
>>> lpr_s_t.set_lower([1,0,0,0,0,0,0,0,0], 0.500)
>>> lpr_s_t.set_upper([1,0,0,0,0,0,0,0,0], 0.666)
>>> lpr_s_t.set_lower([0,1,0,0,0,0,0,0,0], 0.222)
>>> lpr_s_t.set_upper([0,1,0,0,0,0,0,0,0], 0.272)
>>> lpr_s_t.set_lower([0,0,1,0,0,0,0,0,0], 0.125)
>>> lpr_s_t.set_upper([0,0,1,0,0,0,0,0,0], 0.181)
>>> lpr_s_t.set_lower([0,0,0,1,0,0,0,0,0], 0.222)
>>> lpr_s_t.set_upper([0,0,0,1,0,0,0,0,0], 0.333)
>>> lpr_s_t.set_lower([0,0,0,0,1,0,0,0,0], 0.363)
>>> lpr_s_t.set_upper([0,0,0,0,1,0,0,0,0], 0.444)
>>> lpr_s_t.set_lower([0,0,0,0,0,1,0,0,0], 0.250)
>>> lpr_s_t.set_upper([0,0,0,0,0,1,0,0,0], 0.363)
>>> lpr_s_t.set_lower([0,0,0,0,0,0,1,0,0], 0.111)
>>> lpr_s_t.set_upper([0,0,0,0,0,0,1,0,0], 0.166)
>>> lpr_s_t.set_lower([0,0,0,0,0,0,0,1,0], 0.333)
>>> lpr_s_t.set_upper([0,0,0,0,0,0,0,1,0], 0.363)
>>> lpr_s_t.set_lower([0,0,0,0,0,0,0,0,1], 0.454)
>>> lpr_s_t.set_upper([0,0,0,0,0,0,0,0,1], 0.625)
>>> lpr_s_t.is_avoiding_sure_loss()
False
```

3.1.5 Coherence

```
>>> lpr = LowPoly(pspace=4)
>>> lpr.set_lower([1,2,3,0], 2.5)
>>> lpr.is_coherent()
True
>>> lpr.set_upper([2,4,6,0], 3) # coherence requires at least 5
>>> lpr.is_coherent()
False
```

3.1.6 Linearity

```
>>> lpr = LowPoly(pspace=4)
>>> lpr.set_lower([1,2,3,0], 2.5)
>>> lpr.is_linear()
False
>>> lpr.set_upper([2,4,6,0], 5)
>>> lpr.is_linear()
True
```

3.1.7 Marginal Extension

Finding coherent lower and upper bounds for the oil catter example in [2]:

```
>>> # lower previsions over s, given t
>>> lpr_s_t = [LowPoly(pspace=3),
...            LowPoly(pspace=3),
...            LowPoly(pspace=3)]
>>> # lower prevision over t
>>> lpr_t = LowPoly(pspace=3)
>>> lpr_s_t[0].set_lower([1,0,0], 0.5)
>>> lpr_s_t[0].set_upper([1,0,0], 0.666)
>>> lpr_s_t[0].set_lower([0,1,0], 0.222)
>>> lpr_s_t[0].set_upper([0,1,0], 0.272)
>>> lpr_s_t[0].set_lower([0,0,1], 0.125)
>>> lpr_s_t[0].set_upper([0,0,1], 0.181)
>>> lpr_s_t[0].is_coherent()
False
>>> print(float(lpr_s_t[0].get_lower([1,0,0]))) # not coherent!
0.547
>>> print(float(lpr_s_t[0].get_upper([1,0,0]))) # not coherent!
0.653
>>> print(float(lpr_s_t[0].get_lower([0,1,0])))
0.222
>>> print(float(lpr_s_t[0].get_upper([0,1,0])))
0.272
>>> print(float(lpr_s_t[0].get_lower([0,0,1])))
0.125
>>> print(float(lpr_s_t[0].get_upper([0,0,1])))
0.181
>>> lpr_s_t[1].set_lower([1,0,0], 0.222)
>>> lpr_s_t[1].set_upper([1,0,0], 0.333)
>>> lpr_s_t[1].set_lower([0,1,0], 0.363)
>>> lpr_s_t[1].set_upper([0,1,0], 0.444)
>>> lpr_s_t[1].set_lower([0,0,1], 0.250)
>>> lpr_s_t[1].set_upper([0,0,1], 0.363)
>>> lpr_s_t[1].is_coherent()
True
>>> print(float(lpr_s_t[1].get_lower([1,0,0])))
0.222
>>> print(float(lpr_s_t[1].get_upper([1,0,0])))
0.333
>>> print(float(lpr_s_t[1].get_lower([0,1,0])))
0.363
```

[2] Kikuti, D., Cozman, F., de Campos, C.: Partially ordered preferences in decision trees: Computing strategies with imprecision in probabilities. In: R. Brafman, U. Junker (eds.) IJCAI-05 Multidisciplinary Workshop on Advances in Preference Handling, pp. 118–123, 2005.

```
>>> print(float(lpr_s_t[1].get_upper([0,1,0])))
0.444
>>> print(float(lpr_s_t[1].get_lower([0,0,1])))
0.25
>>> print(float(lpr_s_t[1].get_upper([0,0,1])))
0.363
>>> lpr_s_t[2].set_lower([1,0,0], 0.111)
>>> lpr_s_t[2].set_upper([1,0,0], 0.166)
>>> lpr_s_t[2].set_lower([0,1,0], 0.333)
>>> lpr_s_t[2].set_upper([0,1,0], 0.363)
>>> lpr_s_t[2].set_lower([0,0,1], 0.454)
>>> lpr_s_t[2].set_upper([0,0,1], 0.625)
>>> lpr_s_t[2].is_coherent()
False
>>> print(float(lpr_s_t[2].get_lower([1,0,0])))
0.111
>>> print(float(lpr_s_t[2].get_upper([1,0,0])))
0.166
>>> print(float(lpr_s_t[2].get_lower([0,1,0])))
0.333
>>> print(float(lpr_s_t[2].get_upper([0,1,0])))
0.363
>>> print(float(lpr_s_t[2].get_lower([0,0,1]))) # not coherent!
0.471
>>> print(float(lpr_s_t[2].get_upper([0,0,1]))) # not coherent!
0.556
>>> lpr_t.set_lower([1,0,0], 0.181)
>>> lpr_t.set_upper([1,0,0], 0.222)
>>> lpr_t.set_lower([0,1,0], 0.333)
>>> lpr_t.set_upper([0,1,0], 0.363)
>>> lpr_t.set_lower([0,0,1], 0.444)
>>> lpr_t.set_upper([0,0,1], 0.454)
>>> lpr_t.is_coherent()
False
>>> print(float(lpr_t.get_lower([1,0,0]))) # not coherent!
0.183
>>> print(float(lpr_t.get_upper([1,0,0])))
0.222
>>> print(float(lpr_t.get_lower([0,1,0])))
0.333
>>> print(float(lpr_t.get_upper([0,1,0])))
0.363
>>> print(float(lpr_t.get_lower([0,0,1])))
0.444
>>> print(float(lpr_t.get_upper([0,0,1])))
0.454
>>> # now some calculations
>>> gamble_s = [-7, 5, 20]
>>> minus_gamble_s = [7, -5, -20]
>>> print(float(lpr_s_t[0].get_lower(gamble_s)))
-0.961
>>> print(float(lpr_s_t[0].get_lower(minus_gamble_s)))
-1.151
>>> print(float(lpr_s_t[1].get_lower(gamble_s)))
4.754
>>> print(float(lpr_s_t[1].get_lower(minus_gamble_s)))
-7.781
>>> print(float(lpr_s_t[2].get_lower(gamble_s)))
```

```
10.073
>>> print(float(lpr_s_t[2].get_lower(minus_gamble_s)))
-12.008
>>> # calculate its lower prevision by marginal extension
>>> # XXX this is a quick hackish way to set up the marginal extension
>>> # XXX see decision tree example for something more sane
>>> lpr = LowPoly(pspace=9)
>>> def get_lower(gamble_s_t):
...     return lpr_t.get_lower([lpr_s.get_lower(gamble_s)
...                             for lpr_s, gamble_s
...                             in zip(lpr_s_t, gamble_s_t)])
>>> def get_upper(gamble_s_t):
...     return lpr_t.get_upper([lpr_s.get_upper(gamble_s)
...                             for lpr_s, gamble_s
...                             in zip(lpr_s_t, gamble_s_t)])
>>> lpr.get_lower = get_lower
>>> lpr.get_upper = get_upper
>>> # a gamble which is a function of s and t; t comes as first
>>> # argument (in order to match lpr_s_t): gamble_s_t[t][s]
>>> gamble_s_t = [[-7,5,20], [-7,5,20], [-7,5,20]]
>>> # do the calculations
>>> print(float(lpr.get_lower(gamble_s_t)))
5.846906
>>> print(float(lpr.get_upper(gamble_s_t)))
8.486768
>>> # another gamble which is a function of s and t
>>> gamble_s_t = [[-7,5,20], [0,0,0], [0,0,0]]
>>> # calculate its lower prevision by marginal extension
>>> print(float(lpr.get_lower(gamble_s_t)))
-0.213342
>>> # another gamble which is a function of s and t
>>> gamble_s_t = [[7,-5,-20], [0,0,0], [0,0,0]]
>>> # calculate its lower prevision by marginal extension
>>> print(float(lpr.get_lower(gamble_s_t)))
-0.255522
>>> # another gamble which is a function of s and t
>>> gamble_s_t = [[-1,-1,-1], [-1,11,26], [-1,11,26]]
>>> # calculate its lower prevision by marginal extension
>>> print(float(lpr.get_lower(gamble_s_t)))
10.506248
>>> print(float(lpr.get_upper(gamble_s_t)))
12.995135
```

3.1.8 Generalized Bayes Rule

The next example is effectively the Monty Hall problem.

```
>>> lpr = LowPoly(pspace=4, number_type='fraction')
>>> lpr.set_lower([1, 1, 0, 0], '1/3')
>>> lpr.set_lower([0, 0, 1, 0], '1/3')
>>> lpr.set_lower([0, 0, 0, 1], '1/3')
>>> print(lpr.get_lower([1, 1, 0, 0], set([0, 3])))
0
>>> print(lpr.get_lower([0, 0, 1, 1], set([0, 3])))
1/2
```

3.2 Lower Probabilities

class `improb.lowprev.lowprob.`**`LowProb`**(*pspace=None*, *mapping=None*, *lprev=None*, *uprev=None*, *prev=None*, *lprob=None*, *uprob=None*, *prob=None*, *bba=None*, *credalset=None*, *number_type=None*)

Bases: `improb.lowprev.lowpoly.LowPoly`

An unconditional lower probability. This class is identical to `LowPoly`, except that only unconditional assessments on events are allowed.

```
>>> print(LowProb(3, lprob={(0, 1): '0.1', (1, 2): '0.2'}))
0 1   : 1/10
  1 2 : 1/5
>>> print(LowProb(3, lprev={(3, 1, 0): 1}))
Traceback (most recent call last):
    ...
ValueError: not an indicator gamble
>>> print(LowProb(3, uprob={(0, 1): '0.1'}))
Traceback (most recent call last):
    ...
ValueError: cannot specify upper prevision
>>> print(LowProb(3, mapping={((3, 1, 0), (0, 1)): (1.4, None)}))
Traceback (most recent call last):
    ...
ValueError: not unconditional
>>> lpr = LowProb(3, lprob={(0, 1): '0.1', (1, 2): '0.2', (2,): '0.05'})
>>> lpr.extend()
>>> print(lpr)
      : 0
0     : 0
  1   : 0
    2 : 1/20
0 1   : 1/10
0   2 : 1/20
  1 2 : 1/5
0 1 2 : 1
>>> print(lpr.mobius)
      : 0
0     : 0
  1   : 0
    2 : 1/20
0 1   : 1/10
0   2 : 0
  1 2 : 3/20
0 1 2 : 7/10
```

classmethod **`get_constraints_n_monotone`**(*pspace*, *monotonicity=None*)

Yields constraints for lower probabilities with given monotonicity.

Parameters

- **pspace** (`list` or similar; see *Possibility Spaces*) – The possibility space.
- **monotonicity** (`int` or `collections.Iterable` of `int`) – Requested level of monotonicity (see notes below for details).

As described in `get_constraints_bba_n_monotone()`, the n-monotonicity constraints on basic belief assignment are:

$$\sum_{B\colon C\subseteq B\subseteq A} m(B) \geq 0$$

for all $C \subseteq A \subseteq \Omega$, with $1 \leq |C| \leq n$.

By the Mobius transform, this is equivalent to:

$$\sum_{B\colon C\subseteq B\subseteq A} \sum_{D\colon D\subseteq B} (-1)^{|B\setminus D|} \underline{P}(D) \geq 0$$

Once noted that

$$(C \subseteq B \subseteq A \quad \& \quad D \subseteq B) \iff (C \cup D \subseteq B \subseteq A \quad \& \quad D \subseteq A),$$

we can conveniently rewrite the sum as:

$$\sum_{D\colon D\subseteq A} \sum_{B\colon C\cup D\subseteq B\subseteq A} (-1)^{|B\setminus D|} \underline{P}(D) \geq 0$$

This implementation iterates over all $C \subseteq A \subseteq \Omega$, with $|C| = n$, and yields each constraint as an iterable of (event, coefficient) pairs, where zero coefficients are omitted.

Note: As just mentioned, this method returns the constraints corresponding to the latter equation for $|C|$ equal to *monotonicity*. To get all the constraints for n-monotonicity, call this method with *monotonicity=xrange(1, n + 1)*.

The rationale for this approach is that, in case you already know that (n-1)-monotonicity is satisfied, then you only need the constraints for *monotonicity=n* to check for n-monotonicity.

Note: The trivial constraints that the empty set must have lower probability zero, and that the possibility space must have lower probability one, are not included: so for *monotonicity=0* this method returns an empty iterator.

```
>>> pspace = PSpace("abc")
>>> for mono in xrange(1, len(pspace) + 1):
...     print("{0} monotonicity:".format(mono))
...     print(" ".join("{0:<{1}}".format("".join(i for i in event), len(pspace))
...                    for event in pspace.subsets()))
...     constraints = [
...         dict(constraint) for constraint in
...         LowProb.get_constraints_n_monotone(pspace, mono)]
...     constraints = [
...         [constraint.get(event, 0) for event in pspace.subsets()]
...         for constraint in constraints]
...     for constraint in sorted(constraints):
...         print(" ".join("{0:<{1}}".format(value, len(pspace))
...                        for value in constraint))
...
1 monotonicity:
    a   b   c   ab  ac  bc  abc
-1  0   0   1   0   0   0   0
-1  0   1   0   0   0   0   0
-1  1   0   0   0   0   0   0
0   -1  0   0   0   1   0   0
0   -1  0   0   1   0   0   0
0   0   -1  0   0   0   1   0
0   0   -1  0   1   0   0   0
0   0   0   -1  0   0   1   0
0   0   0   -1  0   1   0   0
```

```
0    0    0    0    -1   0    0    1
0    0    0    0    0    -1   0    1
0    0    0    0    0    0    -1   1
2 monotonicity:
     a    b    c    ab   ac   bc   abc
0    0    0    1    0    -1   -1   1
0    0    1    0    -1   0    -1   1
0    1    0    0    -1   -1   0    1
1    -1   -1   0    1    0    0    0
1    -1   0    -1   0    1    0    0
1    0    -1   -1   0    0    1    0
3 monotonicity:
     a    b    c    ab   ac   bc   abc
-1   1    1    1    -1   -1   -1   1
```

is_completely_monotone()

Checks whether the lower probability is completely monotone or not.

> **Warning:** The lower probability must be defined for all events. If needed, call `extend()` first.

```
>>> lpr = LowProb(
...     pspace='abcd',
...     lprob={'ab': '0.2', 'bc': '0.2', 'abc': '0.2', 'b': '0.1'})
>>> lpr.extend()
>>> print(lpr)
        : 0
a       : 0
  b     : 1/10
    c   : 0
      d : 0
a b     : 1/5
a   c   : 0
a     d : 0
  b c   : 1/5
  b   d : 1/10
    c d : 0
a b c   : 1/5
a b   d : 1/5
a   c d : 0
  b c d : 1/5
a b c d : 1
>>> print(lpr.mobius)
        : 0
a       : 0
  b     : 1/10
    c   : 0
      d : 0
a b     : 1/10
a   c   : 0
a     d : 0
  b c   : 1/10
  b   d : 0
    c d : 0
a b c   : -1/10
a b   d : 0
a   c d : 0
  b c d : 0
```

```
a b c d : 4/5
>>> lpr.is_completely_monotone() # (it is in fact not even 2-monotone)
False

>>> lpr = LowProb(
...     pspace='abcd',
...     lprob={'ab': '0.2', 'bc': '0.2', 'abc': '0.3', 'b': '0.1'})
>>> lpr.extend()
>>> print(lpr)
        : 0
a       : 0
  b     : 1/10
    c   : 0
      d : 0
a b     : 1/5
a   c   : 0
a     d : 0
  b c   : 1/5
  b   d : 1/10
    c d : 0
a b c   : 3/10
a b   d : 1/5
a   c d : 0
  b c d : 1/5
a b c d : 1
>>> print(lpr.mobius)
        : 0
a       : 0
  b     : 1/10
    c   : 0
      d : 0
a b     : 1/10
a   c   : 0
a     d : 0
  b c   : 1/10
  b   d : 0
    c d : 0
a b c   : 0
a b   d : 0
a   c d : 0
  b c d : 0
a b c d : 7/10
>>> lpr.is_completely_monotone()
True
```

is_n_monotone(*monotonicity=None*)

Given that the lower probability is (n-1)-monotone, is the lower probability n-monotone?

Note: To check for n-monotonicity, call this method with *monotonicity=xrange(n + 1)*.

Note: For convenience, 0-montonicity is defined as empty set and possibility space having lower probability 0 and 1 respectively.

Warning: The lower probability must be defined for all events. If needed, call `extend()` first.

Warning: For large levels of monotonicity, it is slightly more efficient to call `is_bba_n_monotone()` on `mobius`.

classmethod `make_extreme_n_monotone` (*pspace*, *monotonicity=None*)

Yield extreme lower probabilities with given monotonicity.

> **Warning:** Currently this doesn't work very well except for the cases below.

```
>>> lprs = list(LowProb.make_extreme_n_monotone('abc', monotonicity=2))
>>> len(lprs)
8
>>> all(lpr.is_coherent() for lpr in lprs)
True
>>> all(lpr.is_n_monotone(2) for lpr in lprs)
True
>>> all(lpr.is_n_monotone(3) for lpr in lprs)
False
>>> lprs = list(LowProb.make_extreme_n_monotone('abc', monotonicity=3))
>>> len(lprs)
7
>>> all(lpr.is_coherent() for lpr in lprs)
True
>>> all(lpr.is_n_monotone(2) for lpr in lprs)
True
>>> all(lpr.is_n_monotone(3) for lpr in lprs)
True
>>> lprs = list(LowProb.make_extreme_n_monotone('abcd', monotonicity=2))
>>> len(lprs)
41
>>> all(lpr.is_coherent() for lpr in lprs)
True
>>> all(lpr.is_n_monotone(2) for lpr in lprs)
True
>>> all(lpr.is_n_monotone(3) for lpr in lprs)
False
>>> all(lpr.is_n_monotone(4) for lpr in lprs)
False
>>> lprs = list(LowProb.make_extreme_n_monotone('abcd', monotonicity=3))
>>> len(lprs)
16
>>> all(lpr.is_coherent() for lpr in lprs)
True
>>> all(lpr.is_n_monotone(2) for lpr in lprs)
True
>>> all(lpr.is_n_monotone(3) for lpr in lprs)
True
>>> all(lpr.is_n_monotone(4) for lpr in lprs)
False
>>> lprs = list(LowProb.make_extreme_n_monotone('abcd', monotonicity=4))
>>> len(lprs)
15
>>> all(lpr.is_coherent() for lpr in lprs)
True
>>> all(lpr.is_n_monotone(2) for lpr in lprs)
True
>>> all(lpr.is_n_monotone(3) for lpr in lprs)
True
>>> all(lpr.is_n_monotone(4) for lpr in lprs)
True
>>> # cddlib hangs on larger possibility spaces
```

```
>>> #lprs = list(LowProb.make_extreme_n_monotone('abcde', monotonicity=2))
```

classmethod make_random (*pspace=None, division=None, zero=True, number_type='float'*)
Generate a random coherent lower probability.

mobius
The mobius transform of the assigned unconditional lower probabilities, as `SetFunction`.

See Also:

`improb.setfunction.SetFunction.get_mobius()` Mobius transform calculation of an arbitrary set function.

`improb.lowprev.belfunc.BelFunc` Belief functions.

set_function
The lower probability as `SetFunction`.

3.3 Belief Functions

class `improb.lowprev.belfunc.`**BelFunc** (*pspace=None, mapping=None, lprev=None, uprev=None, prev=None, lprob=None, uprob=None, prob=None, bba=None, credalset=None, number_type=None*)
Bases: `improb.lowprev.lowprob.LowProb`

This identical to `LowProb`, except that it uses the Mobius transform to calculate the natural extension.

get_lower (*gamble, event=True, algorithm='mobius'*)
Calculate the lower expectation of a gamble.

The default algorithm is to use the Mobius transform m of the lower probability $\underline{P}$:

$$\underline{E}(f) = \sum_{A\subseteq\Omega} m(A) \inf_{\omega\in A} f(\omega)$$

See Also:

`improb.lowprev.lowprob.LowProb.is_completely_monotone()` To check for complete monotonicity.

`improb.setfunction.SetFunction.get_mobius()` Mobius transform of an arbitrary set function.

Warning: To use the Mobius transform, the domain of the lower probability must contain *all* events. If needed, call `extend()`:

```
>>> bel = BelFunc(2, lprob=['0.2', '0.25'])
>>> print(bel)
0   : 1/5
  1 : 1/4
>>> bel.get_lower([1, 3]) # oops! fails...
Traceback (most recent call last):
    ...
KeyError: ...
>>> # solve linear program instead of trying Mobius transform
>>> bel.get_lower([1, 3], algorithm='linprog') # 1 * 0.75 + 3 * 0.25 = 1.5
Fraction(3, 2)
>>> bel.extend()
>>> print(bel)
    : 0
0   : 1/5
  1 : 1/4
0 1 : 1
>>> # now try with Mobius transform; should give same result
>>> bel.get_lower([1, 3]) # now it works
Fraction(3, 2)
```

Warning: With the Mobius algorithm, this method will *not* raise an exception even if the assessments are not completely monotone, or even incoherent—the Mobius transform is in such case still defined, although some of the values of m will be negative (obviously, in such case, $\underline{E}$ will be incoherent as well).

```
>>> bel = BelFunc(
...     pspace='abcd',
...     lprob={'ab': '0.2', 'bc': '0.2', 'abc': '0.2', 'b': '0.1'})
>>> bel.extend()
>>> bel.is_completely_monotone() # (it is in fact not even 2-monotone)
False
>>> # exact linear programming algorithm
>>> bel.get_lower([1, 2, 1, 0], algorithm='linprog')
Fraction(2, 5)
>>> # mobius algorithm: different result!!
>>> bel.get_lower([1, 2, 1, 0])
Fraction(3, 10)
```

```
>>> from improb.lowprev.belfunc import BelFunc
>>> from improb.lowprev.lowprob import LowProb
>>> from improb import PSpace
>>> pspace = PSpace(2)
>>> lowprob = LowProb(pspace, lprob=['0.3', '0.2'])
>>> lowprob.extend()
>>> lowprob.is_completely_monotone()
True
>>> print(lowprob)
    : 0
0   : 3/10
  1 : 1/5
0 1 : 1
```

```
>>> print(lowprob.mobius)
    : 0
0   : 3/10
  1 : 1/5
0 1 : 1/2
>>> lpr = BelFunc(pspace, bba=lowprob.mobius)
>>> lpr.is_completely_monotone()
True
>>> print(lpr)
    : 0
0   : 3/10
  1 : 1/5
0 1 : 1
>>> print(lpr.mobius)
    : 0
0   : 3/10
  1 : 1/5
0 1 : 1/2
>>> print(lpr.get_lower([1,0]))
3/10
>>> print(lpr.get_lower([0,1]))
1/5
>>> print(lpr.get_lower([4,9])) # 0.8 * 4 + 0.2 * 9
5
>>> print(lpr.get_lower([5,1])) # 0.3 * 5 + 0.7 * 1
11/5
```

3.4 Linear Vacuous Mixtures

class `improb.lowprev.linvac.`**`LinVac`**(*pspace=None, mapping=None, lprev=None, uprev=None, prev=None, lprob=None, uprob=None, prob=None, bba=None, credalset=None, number_type=None*)

Bases: `improb.lowprev.belfunc.BelFunc`

Linear-vacuous mixture, implemented as an unconditional lower probability on singletons.

```
>>> from improb.lowprev.prob import Prob
>>> lpr = Prob(3, prob=['0.2', '0.3', '0.5']).get_linvac('0.1')
>>> print(lpr.get_lower([1,0,0]))
9/50
>>> print(lpr.get_lower([0,1,0]))
27/100
>>> print(lpr.get_lower([0,0,1]))
9/20
>>> print(lpr.get_lower([3,2,1]))
163/100
>>> print(lpr.get_upper([3,2,1]))
183/100
>>> lpr = Prob(4, prob=['0.42', '0.08', '0.18', '0.32']).get_linvac('0.1')
>>> print(lpr.get_lower([5,5,-5,-5]))
-1/2
>>> print(lpr.get_lower([5,5,-5,-5], set([0,2]))) # (6 - 31 * 0.1) / (3 + 2 * 0.1)
29/32
>>> print(lpr.get_lower([-5,-5,5,5], set([1,3]))) # (6 - 31 * 0.1) / (2 + 3 * 0.1)
29/23
>>> print(lpr.get_lower([0,5,0,-5])) # -(6 + 19 * 0.1) / 5
```

```
-79/50
>>> print(lpr.get_lower([0,-5,0,5])) # (6 - 31 * 0.1) / 5
29/50
```

get_lower (*gamble*, *event=True*, *algorithm='linvac'*)

Calculate the lower expectation of a gamble conditional on an event, by the following formula:

$$\underline{E}(f|A) = \frac{(1-\epsilon)\sum_{\omega\in A} p(\omega)f(\omega) + \epsilon \min_{\omega\in A} f(\omega)}{(1-\epsilon)\sum_{\omega\in A} p(\omega) + \epsilon}$$

where $\epsilon = 1 - \sum_\omega \underline{P}(\omega)$ and $p(\omega) = \underline{P}(\omega)/(1-\epsilon)$. Here, $\underline{P}(\omega)$ is simply:

```
self[{omega: 1}, True][0]
```

This method will *not* raise an exception, even if the assessments are incoherent (obviously, in such case, $\underline{E}$ will be incoherent as well). It will raise an exception if not all lower probabilities on singletons are defined (if needed, extend it first).

3.5 Probability Measures

class improb.lowprev.prob.**Prob** (*pspace=None*, *mapping=None*, *lprev=None*, *uprev=None*, *prev=None*, *lprob=None*, *uprob=None*, *prob=None*, *bba=None*, *credalset=None*, *number_type=None*)

Bases: improb.lowprev.linvac.LinVac

A probability measure.

```
>>> p = Prob(5, prob=['0.1', '0.2', '0.3', '0.05', '0.35'])
>>> print(p)
0 : 1/10
1 : 1/5
2 : 3/10
3 : 1/20
4 : 7/20
>>> print(p.get_precise([2, 4, 3, 8, 1]))
53/20
>>> print(p.get_precise([2, 4, 3, 8, 1], [0, 1]))
10/3
```

get_linvac (*epsilon*)

Convert probability into a linear vacuous mixture:

$$\underline{E}(f) = (1-\epsilon)E(f) + \epsilon \inf f$$

get_precise (*gamble*, *event=True*, *algorithm='linear'*)

Calculate the conditional expectation,

$$E(f|A) = \frac{\sum_{\omega\in A} p(\omega)f(\omega)}{\sum_{\omega\in A} p(\omega)}$$

where $p(\omega)$ is simply the probability of the singleton ω:

```
self[{omega: 1}, True][0]
```

classmethod make_random (*pspace=None*, *division=None*, *zero=True*, *number_type='float'*)

Generate a random probability mass function.

```
>>> import random
>>> random.seed(25)
>>> print(Prob.make_random("abcd", division=10))
a : 0.4
b : 0.0
c : 0.1
d : 0.5
>>> random.seed(25)
>>> print(Prob.make_random("abcd", division=10, zero=False))
a : 0.3
b : 0.1
c : 0.2
d : 0.4
```

3.6 More Examples

3.6.1 Mobius Transform and Natural Extension

This example shows that, in general, the Mobius transform of a coherent lower probability cannot be used to calculate its natural extension.

```
>>> import itertools
>>> import random
>>> from improb.lowprev.lowprob import LowProb
>>> from improb.lowprev.belfunc import BelFunc
>>> random.seed(10)
>>> n = 4
>>> events = [list(event) for event in itertools.product([0, 1], repeat=n)]
>>> gambles = [[random.randint(0,2) for i in range(n)] for j in range(20)]
>>> for i in range(20):
...     # construct belief function from lower probability
...     lpr = LowProb.make_random(pspace=n, division=2, number_type='fraction')
...     lpr.extend()
...     bel = BelFunc(lpr)
...     # check for incoherence
...     for gamble in gambles:
...         if lpr.number_cmp(lpr.get_lower(gamble), bel.get_lower(gamble)) != 0:
...             print('lpr (lower probability):')
...             print(lpr)
...             print('bel.mobius (basic belief assignment):')
...             print(bel.mobius)
...             print("lpr.get_lower({0})={1}".format(gamble, lpr.get_lower(gamble)))
...             print("bel.get_lower({0})={1}".format(gamble, bel.get_lower(gamble)))
...             break
...     else:
...         # no incoherence found! try another one
...         continue
...     break
... else:
...     raise RuntimeError("no counterexample found")
lpr (lower probability):
        : 0
0       : 0
  1     : 0
    2   : 0
```

```
      3 : 0
0 1     : 0
0   2   : 1/2
0     3 : 1/2
  1 2   : 1/2
  1   3 : 1/2
    2 3 : 0
0 1 2   : 1/2
0 1   3 : 1/2
0   2 3 : 1/2
  1 2 3 : 1/2
0 1 2 3 : 1
bel.mobius (basic belief assignment):
        : 0
0       : 0
  1     : 0
    2   : 0
      3 : 0
0 1     : 0
0   2   : 1/2
0     3 : 1/2
  1 2   : 1/2
  1   3 : 1/2
    2 3 : 0
0 1 2   : -1/2
0 1   3 : -1/2
0   2 3 : -1/2
  1 2 3 : -1/2
0 1 2 3 : 1
lpr.get_lower([2, 0, 1, 1])=1
bel.get_lower([2, 0, 1, 1])=1/2
```

Quick check:

```
lpr.get_lower([2,0,1,1]) >= lpr([1,0,1,0])+lpr([1,0,0,1])=1
bel.get_lower([2,0,1,1]) = 0.5*(1+1+0+0+0+0-1+0+0)=0.5
```

However, it seems that the Mobius transform of a 2-monotone lower probability *can* be used to calculate its natural extension. The following simulation seems to confirm this for a space of size 3 (all coherent lower probabilities on such space are 2-monotone).

```
>>> import itertools
>>> import random
>>> from improb.lowprev.lowprob import LowProb
>>> from improb.lowprev.belfunc import BelFunc
>>> random.seed(10)
>>> n = 3
>>> gambles = [[random.randint(0,5) for i in range(n)] for j in range(20)]
>>> for i in range(20): # increase at will...
...     # construct a coherent lower probability
...     lpr = LowProb.make_random(pspace=n, division=2, number_type='fraction')
...     lpr.extend()
...     bel = BelFunc(lpr)
...     # check for incoherence
...     for gamble in gambles:
...         if lpr.number_cmp(lpr.get_lower(gamble), bel.get_lower(gamble)) != 0:
...             print('lpr (lower probability):')
...             print(lpr)
...             print('bel.mobius (basic belief assignment):')
```

```
...             print(bel.mobius)
...             print("lpr.get_lower({0})={1}".format(gamble, lpr.get_lower(gamble)))
...             print("bel.get_lower({0})={1}".format(gamble, bel.get_lower(gamble)))
...             break
...     else:
...         # no incoherence found! try another one
...         continue
...     break
... else:
...     raise RuntimeError("no counterexample found")
Traceback (most recent call last):
    ...
RuntimeError: no counterexample found
```

3.6.2 A Simple Subtree Imperfect Decision Tree

```
>>> from fractions import Fraction
>>> import itertools
>>> import random
>>> from improb.lowprev.linvac import LinVac
>>> from improb.decision.opt import OptLowPrevMax
>>> random.seed(40)
>>> n1 = 2
>>> n2 = 2
>>> ndec = 3
>>> pspace = [tuple(omega)
...           for omega in itertools.product(list(range(n1)), list(range(n2)))]
>>> for i in range(20):
...     # construct a linear vacuous mixture
...     lprob = [0.8 / len(pspace) for omega in pspace]
...     lpr = LinVac(pspace, lprob=lprob) # no need for randomness
...     opt = OptLowPrevMax(lpr)
...     # construct gambles:
...     # gambles[w1][d][w2] gives the gain of the w1-d-w2 path
...     gambles = [[[random.randint(0, 2) + 0.01 * d + w1 * 0.005
...                  for w2 in range(n2)]
...                 for d in range(ndec)]
...                for w1 in range(n1)]
...     # construct strategies:
...     # strats[i][w1] gives decision of i'th strategy, after observing w1
...     strats = [tuple(strat)
...               for strat in itertools.product(list(range(ndec)), repeat=n1)]
...     # construct normal form gambles:
...     # normgambles[strat][omega] gives the gain of the i'th strategy
...     normgambles = dict(
...         (strat, dict(
...             (omega, gambles[omega[0]][strat[omega[0]]][omega[1]])
...             for omega in pspace))
...         for strat in strats)
...     # construct extensive form gambles:
...     # extgambles[w1][d][omega] gives the gain of decision d after observing w1, as a function of
...     extgambles = [[dict((omega, gambles[w1][d][omega[1]] if omega[0] == w1 else 0)
...                         for omega in pspace)
...                    for d in range(ndec)]
...                   for w1 in range(n1)]
...     #print(gambles)
```

```
...     #print(strats)
...     #print([[normgambles[strat][omega] for omega in pspace] for strat in strats])
...     #print([[[extgambles[w1][d][omega] for omega in pspace] for d in range(ndec)] for w1 in range
...     # calculate normal form solution of subtrees after observing w1
...     local = {}
...     for w1 in range(n1):
...         event = set(w for w in pspace if w[0] == w1)
...         local[w1] = list(opt([extgambles[w1][d] for d in range(ndec)], event))
...         #print(w1, event)
...         #print(local[w1])
...     # calculate full solution by combining all local solutions
...     normlocal = set()
...     for localgambles in itertools.product(*[local[w1] for w1 in range(n1)]):
...         #print localgambles
...         normlocal.add(tuple(sum(localgamble[w] for localgamble in localgambles) for w in pspace))
...     #print(normlocal)
...     # calculate full normal form solution
...     norm = opt([normgambles[strat] for strat in strats])
...     norm = set(tuple(normgamble[w] for w in pspace) for normgamble in norm)
...     #print(norm)
...     # calculate corresponding strategies
...     localstrats = set()
...     normstrats = set()
...     for strat in strats:
...         normgamble = tuple(normgambles[strat][omega] for omega in pspace)
...         if normgamble in norm:
...             normstrats.add(strat)
...         if normgamble in normlocal:
...             localstrats.add(strat)
...     # convert normal form to extensive form
...     normstrats2 = set(itertools.product(*[set(strat[w1] for strat in normstrats) for w1 in range
...     # check if solutions differ
...     if localstrats != normstrats:
...         for w1 in range(n1):
...             for d in range(ndec):
...                 print(
...                     "w1={0}, d={1}: ".format(w1, d)
...                     + " ".join("{0:.3f}".format(x) for x in gambles[w1][d]))
...         print("lpr=" + " ".join(str(x) for x in lprob))
...         print(sorted(normstrats))
...         print(sorted(localstrats))
...         #print(sorted(localstrats - normstrats))
...         #print(sorted(normstrats - localstrats)) # should be empty!
...         #print(sorted(normstrats2))
...         break
...     # stronger violation... never occurs??
...     #if localstrats != normstrats2:
... else:
...     raise RuntimeError("no counterexample found")
w1=0, d=0: 1.000 2.000
w1=0, d=1: 0.010 0.010
w1=0, d=2: 2.020 1.020
w1=1, d=0: 0.005 1.005
w1=1, d=1: 2.015 1.015
w1=1, d=2: 0.025 2.025
lpr=0.2 0.2 0.2 0.2
[(0, 1), (2, 1), (2, 2)]
[(0, 1), (0, 2), (2, 1), (2, 2)]
```

3.6.3 Stronger Violation of Subtree Perfectness

Does subtree imperfectness persists after converting the normal form into an extensive form?

```
>>> import itertools
>>> import random
>>> from improb.lowprev.lowprob import LowProb
>>> from improb.decision.opt import OptLowPrevMax
>>> random.seed(10)
>>> n1 = 2
>>> n2 = 3
>>> ndec = 3
>>> pspace = [tuple(omega)
...           for omega in itertools.product(list(range(n1)), list(range(n2)))]
>>> for i in range(20):
...     # construct a lower prevision
...     lpr = LowProb.make_random(pspace=pspace, zero=False)
...     # construct gambles:
...     # gambles[w1][d][w2] gives the gain of the w1-d-w2 path
...     gambles = [[[random.randint(0, 5) for w2 in range(n2)]
...                 for d in range(ndec)]
...                for w1 in range(n1)]
...     # construct strategies:
...     # strats[i][w1] gives decision of i'th strategy, after observing w1
...     strats = [tuple(strat)
...               for strat in itertools.product(list(range(ndec)), repeat=n1)]
...     # construct normal form gambles:
...     # normgambles[strat][omega] gives the gain of the i'th strategy
...     normgambles = dict(
...         (strat, dict(
...             (omega, gambles[omega[0]][strat[omega[0]]][omega[1]])
...             for omega in pspace))
...         for strat in strats)
...     # construct extensive form gambles:
...     # extgambles[w1][d][omega] gives the gain of decision d after observing w1, as a function of
...     extgambles = [[dict((omega, gambles[w1][d][omega[1]] if omega[0] == w1 else 0)
...                         for omega in pspace)
...                    for d in range(ndec)]
...                   for w1 in range(n1)]
...     #print(gambles)
...     #print(strats)
...     #print([[normgambles[strat][omega] for omega in pspace] for strat in strats])
...     #print([[[extgambles[w1][d][omega] for omega in pspace] for d in range(ndec)] for w1 in range
...     # calculate normal form solution of subtrees after observing w1
...     local = {}
...     for w1 in range(n1):
...         event = set(w for w in pspace if w[0] == w1)
...         local[w1] = list(opt([extgambles[w1][d] for d in range(ndec)], event))
...         #print(w1, event)
...         #print(local[w1])
...     # calculate full solution by combining all local solutions
...     normlocal = set()
...     for localgambles in itertools.product(*[local[w1] for w1 in range(n1)]):
...         #print localgambles
...         normlocal.add(tuple(sum(localgamble[w] for localgamble in localgambles) for w in pspace))
...     #print(normlocal)
...     # calculate full normal form solution
...     norm = opt([normgambles[strat] for strat in strats])
...     norm = set(tuple(normgamble[w] for w in pspace) for normgamble in norm)
```

```
...     #print(norm)
...     # calculate corresponding strategies
...     localstrats = set()
...     normstrats = set()
...     for strat in strats:
...         normgamble = tuple(normgambles[strat][omega] for omega in pspace)
...         if normgamble in norm:
...             normstrats.add(strat)
...         if normgamble in normlocal:
...             localstrats.add(strat)
...     # convert normal form to extensive form
...     normstrats2 = set(itertools.product(*[set(strat[w1] for strat in normstrats) for w1 in range
...     # check if solutions differ
...     # stronger violation!!
...     if localstrats != normstrats2:
...         for w1 in range(n1):
...             for d in range(ndec):
...                 print(
...                     "w1={0}, d={1}: ".format(w1, d)
...                     + " ".join("{0:.2f}".format(x) for x in gambles[w1][d]))
...         for (gamble, event), (lprev, uprev) in lpr.iteritems():
...             print("lpr({0})={1:.2f}".format(gamble, lprev))
...         print(sorted(normstrats2))
...         print(sorted(localstrats))
...         break
... else:
...     raise RuntimeError("no counterexample found")
Traceback (most recent call last):
    ...
RuntimeError: no counterexample found
```

class `improb.lowprev.`**`LowPrev`**

Abstract base class for working with arbitrary lower previsions.

`dominates`(*gamble*, *other_gamble*, *event=True*, *algorithm=None*)

Does *gamble* dominate *other_gamble* in lower prevision?

Parameters

- **gamble** (`dict` or similar; see *Gambles*) – The left hand side gamble.
- **other_gamble** (`dict` or similar; see *Gambles*) – The right hand side gamble.
- **event** (`list` or similar; see *Events*) – The event to condition on.
- **algorithm** (`str`) – The algorithm to use (the default value uses the most efficient algorithm).

Returns `True` if *gamble* dominates *other_gamble*, `False` otherwise.

Return type `bool`

`get_lower`(*gamble*, *event=True*, *algorithm=None*)

Return the lower expectation for *gamble* conditional on *event* via natural extension.

Parameters

- **gamble** (`dict` or similar; see *Gambles*) – The gamble whose upper expectation to find.
- **event** (`list` or similar; see *Events*) – The event to condition on.
- **algorithm** (`str`) – The algorithm to use (the default value uses the most efficient algorithm).

Returns The lower bound for this expectation, i.e. the natural extension of the gamble.

Return type `float` or `Fraction`

get_upper(*gamble*, *event=True*, *algorithm=None*)

Return the upper expectation for *gamble* conditional on *event* via natural extension.

Parameters

- **gamble** (`dict` or similar; see *Gambles*) – The gamble whose upper expectation to find.
- **event** (`list` or similar; see *Events*) – The event to condition on.
- **algorithm** (`str`) – The algorithm to use (`None` for the most efficient algorithm).

Returns The upper bound for this expectation, i.e. the natural extension of the gamble.

Return type `float` or `Fraction`

is_avoiding_sure_loss(*algorithm=None*)

No Dutch book? Does the lower prevision avoid sure loss?

Returns `True` if avoids sure loss, `False` otherwise.

Return type `bool`

is_coherent(*algorithm=None*)

Do all assessments coincide with their natural extension? Is the lower prevision coherent?

Parameters

- **algorithm** (`str`) – The algorithm to use (the default value uses the most efficient algorithm).

Returns `True` if coherent, `False` otherwise.

Return type `bool`

is_linear(*algorithm=None*)

Is the lower prevision a linear prevision? More precisely, we check that the natural extension is linear on the linear span of the domain of the lower prevision.

Parameters

- **algorithm** (`str`) – The algorithm to use (the default value uses the most efficient algorithm).

Returns `True` if linear, `False` otherwise.

Return type `bool`

pspace

An `PSpace` representing the possibility space.

CHAPTER

FOUR

SET FUNCTIONS

class `improb.setfunction.`**`SetFunction`**(*pspace*, *data=None*, *number_type=None*)

A real-valued set function defined on the power set of a possibility space.

Bases: `collections.MutableMapping`, `cdd.NumberTypeable`

`__init__`(*pspace*, *data=None*, *number_type=None*)

Construct a set function on the power set of the given possibility space.

Parameters

- **pspace** (`list` or similar; see *Possibility Spaces*) – The possibility space.
- **data** (`dict`) – A mapping that defines the value on each event (missing values default to zero).

`__repr__`()

```
>>> SetFunction(pspace=3, data={(): 1, (0, 2): 2.1, (0, 1, 2): '1/3'})
SetFunction(pspace=PSpace(3),
            data={(): 1.0,
                  (0, 2): 2.1000000000000001,
                  (0, 1, 2): 0.33333333333333331},
            number_type='float')
>>> SetFunction(pspace=3, data={(): '1.0', (0, 2): '2.1', (0, 1, 2): '1/3'})
SetFunction(pspace=PSpace(3),
            data={(): 1,
                  (0, 2): '21/10',
                  (0, 1, 2): '1/3'},
            number_type='fraction')
```

`__str__`()

```
>>> print(SetFunction(pspace='abc', data={'': '1', 'ac': '2', 'abc': '3.1'}))
      : 1
a   c : 2
a b c : 31/10
```

classmethod **`get_constraints_bba_n_monotone`**(*pspace*, *monotonicity=None*)

Yields constraints for basic belief assignments with given monotonicity.

Parameters

- **pspace** (`list` or similar; see *Possibility Spaces*) – The possibility space.

- **monotonicity** (`int` or `collections.Iterable` of `int`) – Requested level of monotonicity (see notes below for details).

This follows the algorithm described in Proposition 2 (for 1-monotonicity) and Proposition 4 (for n-monotonicity) of *Chateauneuf and Jaffray, 1989. Some characterizations of lower probabilities and other monotone capacities through the use of Mobius inversion. Mathematical Social Sciences 17(3), pages 263-283*:

A set function s defined on the power set of Ω is n-monotone if and only if its Mobius transform m satisfies:

$$m(\emptyset)=0, \qquad \sum_{A\subseteq\Omega} m(A)=1,$$

and

$$\sum_{B\colon C\subseteq B\subseteq A} m(B)\geq 0$$

for all $C\subseteq A\subseteq\Omega$, with $1\leq|C|\leq n$.

This implementation iterates over all $C\subseteq A\subseteq\Omega$, with $|C|=n$, and yields each constraint as an iterable of the events $\{B\colon C\subseteq B\subseteq A\}$. For example, you can then check the constraint by summing over this iterable.

Note: As just mentioned, this method returns the constraints corresponding to the latter equation for $|C|$ equal to *monotonicity*. To get all the constraints for n-monotonicity, call this method with *monotonicity=xrange(1, n + 1)*.

The rationale for this approach is that, in case you already know that (n-1)-monotonicity is satisfied, then you only need the constraints for *monotonicity=n* to check for n-monotonicity.

Note: The trivial constraints that the empty set must have mass zero, and that the masses must sum to one, are not included: so for *monotonicity=0* this method returns an empty iterator.

```
>>> pspace = "abc"
>>> for mono in xrange(1, len(pspace) + 1):
...     print("{0} monotonicity:".format(mono))
...     print(" ".join("{0:<{1}}".format("".join(i for i in event), len(pspace))
...                    for event in PSpace(pspace).subsets()))
...     constraints = SetFunction.get_constraints_bba_n_monotone(pspace, mono)
...     constraints = [set(constraint) for constraint in constraints]
...     constraints = [[1 if event in constraint else 0
...                     for event in PSpace(pspace).subsets()]
...                    for constraint in constraints]
...     for constraint in sorted(constraints):
...         print(" ".join("{0:<{1}}"
...                        .format(value, len(pspace))
...                        for value in constraint))
1 monotonicity:
    a   b   c   ab  ac  bc  abc
0   0   0   1   0   0   0   0
0   0   0   1   0   0   1   0
0   0   0   1   0   1   0   0
0   0   0   1   0   1   1   1
0   0   1   0   0   0   0   0
0   0   1   0   0   0   1   0
0   0   1   0   1   0   0   0
0   0   1   0   1   0   1   1
0   1   0   0   0   0   0   0
0   1   0   0   0   1   0   0
```

```
0   1   0   0   1   0   0   0
0   1   0   0   1   1   0   1
2 monotonicity:
    a   b   c   ab  ac  bc  abc
0   0   0   0   0   0   1   0
0   0   0   0   0   0   1   1
0   0   0   0   0   1   0   0
0   0   0   0   0   1   0   1
0   0   0   0   1   0   0   0
0   0   0   0   1   0   0   1
3 monotonicity:
    a   b   c   ab  ac  bc  abc
0   0   0   0   0   0   0   1
```

get_mobius (*event*)

Calculate the value of the Mobius transform of the given event. The Mobius transform of a set function s is given by the formula:

$$m(A)=\sum_{B\subseteq A}(-1)^{|A\setminus B|}s(B)$$

for any event A.

> **Warning:** The set function must be defined for all subsets of the given event.

```
>>> setfunc = SetFunction(pspace='ab', data={'': 0, 'a': 0.25, 'b': 0.3, 'ab': 1})
>>> print(setfunc)
    : 0.0
a   : 0.25
  b : 0.3
a b : 1.0
>>> inv = SetFunction(pspace='ab',
...                   data=dict((event, setfunc.get_mobius(event))
...                        for event in setfunc.pspace.subsets()))
>>> print(inv)
    : 0.0
a   : 0.25
  b : 0.3
a b : 0.45
```

get_zeta (*event*)

Calculate the value of the zeta transform (inverse Mobius transform) of the given event. The zeta transform of a set function m is given by the formula:

$$s(A)=\sum_{B\subseteq A}m(B)$$

for any event A (note that it is usually assumed that $m(\emptyset)=0$).

> **Warning:** The set function must be defined for all subsets of the given event.

```
>>> setfunc = SetFunction(
...     pspace='ab',
...     data={'': 0, 'a': 0.25, 'b': 0.3, 'ab': 0.45})
>>> print(setfunc)
    : 0.0
```

```
a   : 0.25
  b : 0.3
a b : 0.45
>>> inv = SetFunction(pspace='ab',
...                   data=dict((event, setfunc.get_zeta(event))
...                             for event in setfunc.pspace.subsets()))
>>> print(inv)
    : 0.0
a   : 0.25
  b : 0.3
a b : 1.0
```

is_bba_n_monotone(*monotonicity=None*)

Is the set function, as basic belief assignment, n-monotone, given that it is (n-1)-monotone?

Note: To check for n-monotonicity, call this method with *monotonicity=xrange(n + 1)*.

Note: For convenience, 0-montonicity is defined as empty set and possibility space having lower probability 0 and 1 respectively.

Warning: The set function must be defined for all events.

classmethod make_extreme_bba_n_monotone(*pspace*, *monotonicity=None*)

Yield extreme basic belief assignments with given monotonicity.

Warning: Currently this doesn't work very well except for the cases below.

```
>>> bbas = list(SetFunction.make_extreme_bba_n_monotone('abc', monotonicity=2))
>>> len(bbas)
8
>>> all(bba.is_bba_n_monotone(2) for bba in bbas)
True
>>> all(bba.is_bba_n_monotone(3) for bba in bbas)
False
>>> bbas = list(SetFunction.make_extreme_bba_n_monotone('abc', monotonicity=3))
>>> len(bbas)
7
>>> all(bba.is_bba_n_monotone(2) for bba in bbas)
True
>>> all(bba.is_bba_n_monotone(3) for bba in bbas)
True
>>> bbas = list(SetFunction.make_extreme_bba_n_monotone('abcd', monotonicity=2))
>>> len(bbas)
41
>>> all(bba.is_bba_n_monotone(2) for bba in bbas)
True
>>> all(bba.is_bba_n_monotone(3) for bba in bbas)
False
>>> all(bba.is_bba_n_monotone(4) for bba in bbas)
False
>>> bbas = list(SetFunction.make_extreme_bba_n_monotone('abcd', monotonicity=3))
>>> len(bbas)
16
>>> all(bba.is_bba_n_monotone(2) for bba in bbas)
True
>>> all(bba.is_bba_n_monotone(3) for bba in bbas)
True
```

```
>>> all(bba.is_bba_n_monotone(4) for bba in bbas)
False
>>> bbas = list(SetFunction.make_extreme_bba_n_monotone('abcd', monotonicity=4))
>>> len(bbas)
15
>>> all(bba.is_bba_n_monotone(2) for bba in bbas)
True
>>> all(bba.is_bba_n_monotone(3) for bba in bbas)
True
>>> all(bba.is_bba_n_monotone(4) for bba in bbas)
True
>>> # cddlib hangs on larger possibility spaces
>>> #bbas = list(SetFunction.make_extreme_bba_n_monotone('abcde', monotonicity=2))
```

pspace

An PSpace representing the possibility space.

CHAPTER

FIVE

DECISION MAKING

5.1 Optimality Operators

5.1.1 Abstract Operators

class `improb.decision.opt.`**`Opt`**

Abstract base class for optimality operators.

class `improb.decision.opt.`**`OptPartialPreorder`**

Bases: `improb.decision.opt.Opt`

Abstract base class for optimality operators that use a maximality criterion with respect to a partial preordering.

`is_strictly_larger`(*gamble*, *other_gamble*, *event=True*)

Defines the partial ordering.

class `improb.decision.opt.`**`OptTotalPreorder`**

Bases: `improb.decision.opt.Opt`, `cdd.NumberTypeable`

Abstract base class for optimality operators that use a maximality criterion with respect to a total preordering, which is assumed to be represented via real numbers.

`get_value`(*gamble*, *event=True*)

Defines the total order.

Returns The value of the gamble.

Return type `float` or similar; see *Values*

5.1.2 Concrete Operators

class `improb.decision.opt.`**`OptAdmissible`**(*pspace*, *number_type=None*)

Bases: `improb.decision.opt.OptPartialPreorder`, `cdd.NumberTypeable`

Optimality by pointwise dominance.

`is_strictly_larger`(*gamble*, *other_gamble*, *event=True*)

Check for pointwise dominance.

```
>>> opt = OptAdmissible('abc', number_type='fraction')
>>> opt.is_strictly_larger([1, 2, 3], [1, 2, 3])
False
>>> opt.is_strictly_larger([1, 2, 3], [1, 1, 4])
```

```
False
>>> opt.is_strictly_larger([1, 2, 3], [0, 1, 2])
True
>>> opt.is_strictly_larger([1, 2, 3], [2, 3, 4])
False
>>> opt.is_strictly_larger([1, 2, 3], [1, 2, '8/3'])
True
>>> opt.is_strictly_larger([1, 2, 3], [1, 5, 2], event='ac')
True
```

class `improb.decision.opt.`**`OptLowPrevMax`**(*lowprev*)
Bases: `improb.decision.opt.OptPartialPreorder`

Maximality with respect to a lower prevision.

class `improb.decision.opt.`**`OptLowPrevMaxMin`**(*lowprev*)
Bases: `improb.decision.opt.OptTotalPreorder`

Gamma-maximin with respect to a lower prevision.

class `improb.decision.opt.`**`OptLowPrevMaxMax`**(*lowprev*)
Bases: `improb.decision.opt.OptLowPrevMaxMin`

Gamma-maximax with respect to a lower prevision.

class `improb.decision.opt.`**`OptLowPrevMaxHurwicz`**(*lowprev, alpha*)
Bases: `improb.decision.opt.OptLowPrevMaxMin`

Hurwicz with respect to a lower prevision.

class `improb.decision.opt.`**`OptLowPrevMaxInterval`**(*lowprev*)
Bases: `improb.decision.opt.OptLowPrevMax`

Interval dominance with respect to a lower prevision.

5.1.3 Examples

Example taken from [1]:

```
>>> lpr = LowPoly(pspace=2)
>>> lpr.set_lower([1, 0], 0.28)
>>> lpr.set_upper([1, 0], 0.70)
>>> gambles = [[4, 0], [0, 4], [3, 2], [0.5, 3], [2.35, 2.35], [4.1, -0.3]]
>>> opt = OptLowPrevMax(lpr)
>>> list(opt(gambles)) == [[4, 0], [0, 4], [3, 2], [2.35, 2.35]]
True
>>> list(OptLowPrevMaxMin(lpr)(gambles)) == [[2.35, 2.35]]
True
>>> list(OptLowPrevMaxMax(lpr)(gambles)) == [[0, 4]]
True
>>> list(OptLowPrevMaxInterval(lpr)(gambles)) == [[4, 0], [0, 4], [3, 2], [2.35, 2.35], [4.1, -0.3]]
True
```

Another example:

```
>>> lpr = Prob(pspace=4, prob=[0.42, 0.08, 0.18, 0.32]).get_linvac(0.1)
>>> opt = OptLowPrevMax(lpr)
>>> for c in range(3):
```

[1] Matthias C. M. Troffaes. Decision making under uncertainty using imprecise probabilities. International Journal of Approximate Reasoning, 45(1):17-29, May 2007.

```
...     gambles = [[10-c,10-c,15-c,15-c],[10-c,5-c,15-c,20-c],
...                [5-c,10-c,20-c,15-c],[5-c,5-c,20-c,20-c],
...                [10,10,15,15],[5,5,20,20]]
...     print(list(opt(gambles)))
[[10, 5, 15, 20]]
[[9, 4, 14, 19], [10, 10, 15, 15], [5, 5, 20, 20]]
[[10, 10, 15, 15], [5, 5, 20, 20]]
```

5.2 Decision Trees

class improb.decision.tree.**Tree**

Abstract base class for decision trees.

```
>>> pspace = PSpace("AB", "XY")
>>> A = pspace.make_event("A", "XY", name="A")
>>> B = pspace.make_event("B", "XY", name="B")
>>> X = pspace.make_event("AB", "X", name="X")
>>> Y = pspace.make_event("AB", "Y", name="Y")
>>> t1 = Chance(pspace)
>>> t1[A] = '1' # using strings for fractions
>>> t1[B] = '2/11'
>>> t2 = Chance(pspace)
>>> t2[A] = '5/3'
>>> t2[B] = '6'
>>> t12 = Decision()
>>> t12["d1"] = t1
>>> t12["d2"] = t2
>>> t3 = Chance(pspace)
>>> t3[A] = '8'
>>> t3[B] = '4.5'
>>> t = Chance(pspace)
>>> t[X] = t12
>>> t[Y] = t3
>>> print(t)
O--X--#--d1--O--A--:1
   |  |      |
   |  |      B--:2/11
   |  |
   |  d2--O--A--:5/3
   |         |
   |         B--:6
   |
   Y--O--A--:8
      |
      B--:9/2
>>> t.pspace
PSpace([('A', 'X'), ('A', 'Y'), ('B', 'X'), ('B', 'Y')])
>>> for gamble, normal_tree in t.get_normal_form():
...     print(gamble)
...     print('')
('A', 'X') : 1
('A', 'Y') : 8
('B', 'X') : 2/11
('B', 'Y') : 9/2
<BLANKLINE>
('A', 'X') : 5/3
```

```
('A', 'Y') : 8
('B', 'X') : 6
('B', 'Y') : 9/2
<BLANKLINE>
>>> for gamble, normal_tree in t.get_normal_form():
...     print(normal_tree)
...     print('')
O--X--#--d1--O--A--:1
   |            |
   |            B--:2/11
   |
   Y--O--A--:8
         |
         B--:9/2
<BLANKLINE>
O--X--#--d2--O--A--:5/3
   |            |
   |            B--:6
   |
   Y--O--A--:8
         |
         B--:9/2
<BLANKLINE>
```

_get_norm_back_opt (*opt=None*, *event=True*)

Like `get_norm_back_opt()` but without applying *opt* at the root of the tree in the final stage.

All other normal form methods (`get_normal_form()`, `get_norm_opt()`, and `get_norm_back_opt()`) are defined in terms of this method, so subclasses only need to implement this one as far as normal form calculations are concerned.

__add__ (*value*)

Add a value to all final reward nodes.

Parameters

- **value** (`float` or similar; see *Values*) – The value to add.

__sub__ (*value*)

Subtract a value from all final reward nodes.

Parameters

- **value** (`float` or similar; see *Values*) – The value to subtract.

check_pspace ()

Check the possibility spaces.

Raise `ValueError` on mismatch

get_norm_back_opt (*opt=None*, *event=True*)

Like `get_norm_opt()`, but uses normal form backward induction, which is more efficient.

> **Warning:** If *opt* does not satisfy certain properties, the result can be different from `get_norm_opt()`.

get_norm_opt (*opt=None*, *event=True*)

Get the optimal normal form decisions with respect to the optimality operator *opt*, conditional on *event*. This method does not use backward induction: it simply calculates all normal form decisions and then applies *opt* on them.

Parameters

- **opt** (Opt) – The optimality operator (optional).
- **event** (list or similar; see *Events*) – The event to condition on (optional).

Returns Optimal normal form decisions.

Return type Yields (Gamble, Tree) pairs, where the tree is a normal form decision (i.e. a tree where each decision node has a single branch), and the gamble is the one induced by this tree.

get_normal_form()

Calculate all normal form decisions, and their corresponding gambles.

Returns The normal form of the decision tree.

Return type Yields (Gamble, Tree) pairs, where the tree is a normal form decision (i.e. a tree where each decision node has a single branch), and the gamble is the one induced by this tree.

get_number_type()

Get the number type of the first reward node in the tree.

Returns The number type.

Return type str

pspace

The possibility space, or None if there are no chance nodes in the tree.

class improb.decision.tree.**Reward**(*reward*, *number_type=None*)

Bases: improb.decision.tree.Tree, cdd.NumberTypeable

A reward node.

Parameters

- **reward** (float or similar; see *Values*) – The reward.
- **number_type** (str) – The number type (optional). If omitted, get_number_type_from_value() is used.

```
>>> t = Reward(5)
>>> print(t.pspace)
None
>>> print(t)
:5.0
>>> list(t.get_normal_form())
[(5.0, Reward(5.0, number_type='float'))]
```

class improb.decision.tree.**Decision**(*data=None*)

Bases: improb.decision.tree.Tree

A decision tree rooted at a decision node.

Parameters

- **data** (collections.Mapping) – Mapping from decisions (i.e. strings, but any immutable object would work) to trees (optional).

```
>>> t = Decision({"d1": 5,
...               "d2": 6})
>>> print(t.pspace)
None
```

```
>>> print(t) # dict can change ordering
#--d2--:6.0
   |
   d1--:5.0
>>> for gamble, normal_tree in sorted(t.get_normal_form()):
...     print(gamble)
5.0
6.0
>>> for gamble, normal_tree in sorted(t.get_normal_form()):
...     print(normal_tree)
#--d1--:5.0
#--d2--:6.0
```

class `improb.decision.tree.`**`Chance`**(*pspace*, *data=None*)

Bases: `improb.decision.tree.Tree`

A decision tree rooted at a chance node.

Parameters

- **pspace** (`list` or similar; see *Possibility Spaces*) – The possibility space.
- **data** (`collections.Mapping`) – Mapping from events to trees (optional).

```
>>> t = Chance(pspace=(0, 1), data={(0,): 5, (1,): 6})
>>> t.pspace
PSpace(2)
>>> t.get_number_type()
'float'
>>> print(t)
O--(0)--:5.0
   |
   (1)--:6.0
>>> list(gamble for gamble, normal_tree in t.get_normal_form())
[Gamble(pspace=PSpace(2), mapping={0: 5.0, 1: 6.0})]
```

`check_pspace`()

Events of the chance nodes must form the possibility space.

```
>>> t = Chance(pspace='ab', data={'a': 5, 'ab': 6})
>>> t.check_pspace()
Traceback (most recent call last):
    ...
ValueError: ...
>>> t = Chance(pspace='ab', data={'a': 5})
>>> t.check_pspace()
Traceback (most recent call last):
    ...
ValueError: ...
>>> t = Chance(pspace='ab', data={'a': 5, 'b': 6})
>>> t.check_pspace()
```

5.2.1 Examples

Solving the decision tree for the oil catter example in [2]:

[2] Kikuti, D., Cozman, F., de Campos, C.: Partially ordered preferences in decision trees: Computing strategies with imprecision in probabilities. In: R. Brafman, U. Junker (eds.) IJCAI-05 Multidisciplinary Workshop on Advances in Preference Handling, pp. 118–123, 2005.

```
>>> # specify the decision tree
>>> pspace = PSpace('SWD', 'NOC') # soak, wet, dry; no, open, closed
>>> S = pspace.make_event('S', 'NOC', name="soak")
>>> W = pspace.make_event('W', 'NOC', name="wet")
>>> D = pspace.make_event('D', 'NOC', name="dry")
>>> N = pspace.make_event('SWD', 'N', name="no")
>>> O = pspace.make_event('SWD', 'O', name="open")
>>> C = pspace.make_event('SWD', 'C', name="closed")
>>> lpr = LowPoly(pspace)
>>> t0 = 0
>>> t1 = Chance(pspace)
>>> t1[S] = 20
>>> t1[W] = 5
>>> t1[D] = -7
>>> t = Decision()
>>> t["not drill"] = t0
>>> t["drill"] = t1
>>> ss = t - 1
>>> s = Chance(pspace)
>>> s[N] = ss
>>> s[O] = ss
>>> s[C] = ss
>>> u = Decision()
>>> u["sounding"] = s
>>> u["no sounding"] = t
>>> print(u)
#--sounding-----O--no------#--not drill--:-1.0
   |               |          |
   |               |          drill------O--soak--:19.0
   |               |                        |
   |               |                        wet---:4.0
   |               |                        |
   |               |                        dry---:-8.0
   |               |
   |               open----#--not drill--:-1.0
   |               |          |
   |               |          drill------O--soak--:19.0
   |               |                        |
   |               |                        wet---:4.0
   |               |                        |
   |               |                        dry---:-8.0
   |               |
   |               closed--#--not drill--:-1.0
   |                          |
   |                          drill------O--soak--:19.0
   |                                        |
   |                                        wet---:4.0
   |                                        |
   |                                        dry---:-8.0
   |
   no sounding--#--not drill--:0.0
                   |
                   drill------O--soak--:20.0
                                 |
                                 wet---:5.0
                                 |
                                 dry---:-7.0
>>> lpr = LowPoly(pspace)
```

```
>>> lpr[N, True] = (0.183, 0.222)
>>> lpr[O, True] = (0.333, 0.363)
>>> lpr[C, True] = (0.444, 0.454)
>>> lpr[D, N] = (0.500, 0.666)
>>> lpr[D, O] = (0.222, 0.333)
>>> lpr[D, C] = (0.111, 0.166)
>>> lpr[W, N] = (0.222, 0.272)
>>> lpr[W, O] = (0.363, 0.444)
>>> lpr[W, C] = (0.333, 0.363)
>>> lpr[S, N] = (0.125, 0.181)
>>> lpr[S, O] = (0.250, 0.363)
>>> lpr[S, C] = (0.454, 0.625)
>>> opt = OptLowPrevMax(lpr)
>>> for gamble, normal_tree in u.get_norm_back_opt(opt):
...     print(normal_tree)
#--no sounding--#--drill--O--soak--:20.0
                             |
                             wet---:5.0
                             |
                             dry---:-7.0
>>> # note: backopt should be normopt for maximality!! let's check...
>>> for gamble, normal_tree in u.get_norm_opt(opt):
...     print(normal_tree)
#--no sounding--#--drill--O--soak--:20.0
                             |
                             wet---:5.0
                             |
                             dry---:-7.0
```

Solving the lake district example:

```
>>> c = 1 # cost of newspaper
>>> pspace = PSpace('rs','RS')
>>> R_ = pspace.make_event('r', 'RS', name='predict rain')
>>> S_ = pspace.make_event('s', 'RS', name='predict sunshine')
>>> R = pspace.make_event('rs', 'R', name='rain')
>>> S = pspace.make_event('rs', 'S', name='sunshine')
>>> t0 = Chance(pspace)
>>> t0[R] = 10
>>> t0[S] = 15
>>> t1 = Chance(pspace)
>>> t1[R] = 5
>>> t1[S] = 20
>>> t = Decision()
>>> t["take waterproof"] = t0
>>> t["no waterproof"] = t1
>>> s = Chance(pspace)
>>> s[R_] = t
>>> s[S_] = t
>>> u = Decision()
>>> u["buy newspaper"] = s - c
>>> u["do not buy"] = t
>>> print(u)
#--buy newspaper--O--predict rain------#--take waterproof--O--rain------:9.0
   |                 |                    |                    |
   |                 |                    |                    sunshine--:14.0
   |                 |                    |
   |                 |                    no waterproof----O--rain------:4.0
```

```
   |                 |                                        |
   |                 |                                        sunshine--:19.0
   |                 |
   |                 predict sunshine--#--take waterproof--O--rain------:9.0
   |                                      |                   |
   |                                      |                   sunshine--:14.0
   |                                      |
   |                                      no waterproof----O--rain------:4.0
   |                                                          |
   |                                                          sunshine--:19.0
   |
   do not buy-----#--take waterproof--O--rain------:10.0
                     |                   |
                     |                   sunshine--:15.0
                     |
                     no waterproof----O--rain------:5.0
                                         |
                                         sunshine--:20.0
>>> lpr = LowPoly(pspace)
>>> lpr[R_ & R, True] = (0.42 * 0.9, None)
>>> lpr[R_ & S, True] = (0.18 * 0.9, None)
>>> lpr[S_ & R, True] = (0.08 * 0.9, None)
>>> lpr[S_ & S, True] = (0.32 * 0.9, None)
>>> for c in [0.579, 0.581, 1.579, 1.581]:
...     print("newspaper cost = {0}".format(c))
...     u["buy newspaper"] = s - c
...     opt = OptLowPrevMax(lpr)
...     for gamble, normal_tree in u.get_norm_back_opt(opt):
...         print(normal_tree)
newspaper cost = 0.579
#--buy newspaper--O--predict rain------#--take waterproof--O--rain------:9.421
                     |                                        |
                     |                                        sunshine--:14.421
                     |
                     predict sunshine--#--no waterproof--O--rain------:4.421
                                                            |
                                                            sunshine--:19.421
newspaper cost = 0.581
#--buy newspaper--O--predict rain------#--take waterproof--O--rain------:9.419
                     |                                        |
                     |                                        sunshine--:14.419
                     |
                     predict sunshine--#--no waterproof--O--rain------:4.419
                                                            |
                                                            sunshine--:19.419
#--do not buy--#--take waterproof--O--rain------:10.0
                                      |
                                      sunshine--:15.0
#--do not buy--#--no waterproof--O--rain------:5.0
                                    |
                                    sunshine--:20.0
newspaper cost = 1.579
#--buy newspaper--O--predict rain------#--take waterproof--O--rain------:8.421
                     |                                        |
                     |                                        sunshine--:13.421
                     |
                     predict sunshine--#--no waterproof--O--rain------:3.421
                                                            |
```

```
                                                                         sunshine--:18.421
#--do not buy--#--take waterproof--O--rain------:10.0
                                      |
                                      sunshine--:15.0
#--do not buy--#--no waterproof--O--rain------:5.0
                                    |
                                    sunshine--:20.0
newspaper cost = 1.581
#--do not buy--#--take waterproof--O--rain------:10.0
                                      |
                                      sunshine--:15.0
#--do not buy--#--no waterproof--O--rain------:5.0
                                    |
                                    sunshine--:20.0
```

PYTHON MODULE INDEX

i

INDEX

Symbols

B

C

D

E

G

I

L

M

O

P

R

S

T