Sébastien Labbé Research Code Reference Manual

Release 0.1

Sébastien Labbé

August 27, 2014

CONTENTS

This is the reference manual for the Sébastien Labbé Research Code extension to the Sage mathematical software system. Sage is free open source math software that supports research and teaching in algebra, geometry, number theory, cryptography, and related areas. Sébastien Labbé Research Code implements digital geometry, combinatorics on words and symbolic dynamical systems simulation code in Sage, via a set of new Python classes. Many of the modules corresponds to research code written for published articles (double square tiles, Christoffel graphs, factor complexity). It is meant to be reused and reusable (full documentation including doctests) but still has not be used for other purposes. Comments are welcome.

To use this module, you need to import it:

```
from slabbe import *
```

Warning: The above line is **mandatory** for any doctest of this reference manual to work.

This reference manual contains many examples that illustrate the usage of slabbe spkg. The examples are all tested with each release of slabbe spkg, and should produce exactly the same output as in this manual, except for line breaks.

CHAPTER

ONE

DIGITAL GEOMETRY

1.1 Discrete Subset

Subsets of ZZ^d with the edge relation +e_i and -e_i.

EXAMPLES:

```
sage: DiscreteSubset(2)
Subset of ZZ^2
sage: DiscreteSubset(4)
Subset of ZZ^4
```

A discrete 2d disk:

```
sage: D = DiscreteSubset(2, lambda (x,y) : x^2 + y^2 < 4)
sage: D.list()
[(0, 0), (0, 1), (0, -1), (1, 0), (-1, 0), (-1, 1), (1, -1), (1, 1), (-1, -1)]
sage: D
Subset of ZZ^2
```

A discrete 3d ball:

```
sage: predicate = lambda (x,y,z) : x^2 + y^2 + z^2 <= 4
sage: D = DiscreteSubset(3, predicate)
sage: D
Subset of ZZ^3
sage: (0,0,0) in D
True
sage: (10,10,10) in D
False
sage: len(D.list())
33
sage: D.plot()    # optional long
```

A discrete 4d hyperplane:

```
sage: predicate = lambda (x,y,z,w) : 0 <= 2*x + 3*y + 4*z + 5*w < 14
sage: D = DiscreteSubset(4, predicate)
sage: D
Subset of ZZ^4
sage: D.an_element()
(0, 0, 0, 0)
```

A 2d discrete box:

```
sage: b = DiscreteBox([-5,5], [-5,5])
sage: b
```

```
Box: [-5, 5] x [-5, 5]
sage: b.plot()       # optional long
```

A 3d discrete box:

```
sage: b = DiscreteBox([-2,2], [-5,5], [-5,5])
sage: b
Box: [-2, 2] x [-5, 5] x [-5, 5]
sage: b.plot()       # optional long
```

The intersection of two discrete objects of the same dimension:

```
sage: circ = DiscreteSubset(2, lambda p: p[0]^2+p[1]^2<=100)
sage: b = DiscreteBox([0,10], [0,10])
sage: I = circ & b
sage: I
Intersection of the following objects:
Subset of ZZ^2
[0, 10] x [0, 10]
sage: I.an_element()
(0, 0)
sage: I.plot()      # optional long
```

A discrete tube (preimage of a discrete box by a matrix):

```
sage: M3to2 = matrix(2,[-sqrt(3),sqrt(3),0,-1,-1,2],ring=RR)/2
sage: M3to2
[-0.866025403784439  0.866025403784439  0.000000000000000]
[-0.500000000000000 -0.500000000000000   1.00000000000000]
sage: tube = DiscreteTube([-5,5],[-5,5], projmat=M3to2)
sage: tube
DiscreteTube: Preimage of [-5, 5] x [-5, 5] by a 2 by 3 matrix
sage: it = iter(tube)
sage: [next(it) for _ in range(4)]
[(0, 0, 0), (1, 0, 0), (0, 0, 1), (0, 0, -1)]
```

TODO:

- Code Complement
- The method projection_matrix should be outside of the class?
- DiscreteTube should have a method projection_matrix
- The user should be able to provide an element to the object or a list of element
- Their should be an input saying whether the object is connected or not and what kind of neighbor connectedness
- When zero is not in self, then the an_element method fails (see below)

```
sage: D = DiscreteSubset(2, lambda (x,y) : 4 < x^2 + y^2 < 25)
sage: D.an_element()
Traceback (most recent call last):
...
AssertionError: an_element method returns an element which is not in self
```

class slabbe.discrete_subset.**DiscreteBox**(**args*)

Bases: slabbe.discrete_subset.DiscreteSubset

Cartesian product of intervals.

INPUT:

•*args - intervals, lists of size two : [min, max]

EXAMPLES:

```
sage: DiscreteBox([-5,5],[-5,5])
Box: [-5, 5] x [-5, 5]

sage: D = DiscreteBox([-3,3],[-3,3],[-3,3],[-3,3])
sage: next(iter(D))
(0, 0, 0, 0)
```

TESTS:

```
sage: d = DiscreteBox([-5,5], [-5,5], [-4,4])
sage: d.edges_iterator().next()
((0, 0, 0), (1, 0, 0))
```

clip(*space=1*)

Return a good clip rectangle for this box.

INPUT:

- `space` – number (default: 1), inner space within the box

EXAMPLES:

```
sage: box = DiscreteBox([-6,6],[-6,6])
sage: box
Box: [-6, 6] x [-6, 6]
sage: box.clip()
[(-5, -5), (5, -5), (5, 5), (-5, 5), (-5, -5)]

sage: box = DiscreteBox([-6,6],[-4,3])
sage: box.clip()
[(-5, -3), (5, -3), (5, 2), (-5, 2), (-5, -3)]
```

class slabbe.discrete_subset.**DiscreteSubset**(*dimension=3*, *predicate=None*, *edge_predicate=None*)

Bases: `sage.structure.sage_object.SageObject`

A subset of ZZ^d.

INPUT:

- `dimension` - integer, dimension of the space
- `predicate` - function ZZ^d -> {False, True}
- `edge_predicate` - function ZZ^d,{-1,0,1}^d -> {False, True}

EXAMPLES:

```
sage: DiscreteSubset(3)
Subset of ZZ^3

sage: p = DiscreteSubset(3, lambda x:True)
sage: p
Subset of ZZ^3

sage: fn = lambda p : p[0]+p[1]<p[2]
sage: p = DiscreteSubset(3, fn)
sage: p
Subset of ZZ^3

sage: F = lambda p: Integers(7)(2*p[0]+5*p[1])
sage: edge_predicate = lambda p,s: F(s) < F(s)
sage: D = DiscreteSubset(3, edge_predicate=edge_predicate)
sage: D
Subset of ZZ^3
```

TESTS:

No edges go outside of the box:

```
sage: B = DiscreteBox([-1,1],[-1,1])
sage: len(list(B.edges_iterator()))
12
sage: sorted(B.edges_iterator())
[((-1, -1), (-1, 0)), ((-1, -1), (0, -1)), ((-1, 0), (-1, 1)),
((-1, 0), (0, 0)), ((-1, 1), (0, 1)), ((0, -1), (0, 0)), ((0, -1),
(1, -1)), ((0, 0), (0, 1)), ((0, 0), (1, 0)), ((0, 1), (1, 1)),
((1, -1), (1, 0)), ((1, 0), (1, 1))]
```

an_element()

Returns an immutable element in self.

EXAMPLES:

```
sage: p = DiscreteSubset(3)
sage: p.an_element()
(0, 0, 0)
sage: p.an_element().is_immutable()
True
```

base_edges()

Return a list of positive canonical vectors.

EXAMPLES:

```
sage: d = DiscreteSubset(2)
sage: d.base_edges()
[(1, 0), (0, 1)]

sage: P = DiscretePlane([3,4,5], 12)
sage: P.base_edges()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

children(*p*)

EXAMPLES:

```
sage: p = DiscretePlane([1,pi,7], 1+pi+7, mu=0)
sage: list(p.children(vector((0,0,0))))
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

connected_component_iterator(*roots=None*)

Return an iterator over the connected component of the root.

INPUT:

- `roots` - list of some elements immutable in self

EXAMPLES:

```
sage: p = DiscreteSubset(3)
sage: root = vector((0,0,0))
sage: root.set_immutable()
sage: it = p.connected_component_iterator(roots=[root])
sage: [next(it) for _ in range(5)]
[(0, 0, 0), (1, 0, 0), (0, 0, 1), (0, 0, -1), (0, -1, 0)]

sage: p = DiscretePlane([1,pi,7], 1+pi+7, mu=0)
sage: root = vector((0,0,0))
sage: root.set_immutable()
sage: it = p.connected_component_iterator(roots=[root])
sage: [next(it) for _ in range(5)]
[(0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1), (-1, 1, 0)]
```

d_neighbors(*p*, *d=2*)

Retourne le voisinage du point p, i.e. les points parmi les 3^d possible qui appartiennent a l'objet discret.

INPUT:

- **p** - un point discret
- **d** - integer (optional, default:2),

OUTPUT:

liste de points

EXAMPLES:

```
sage: p = DiscretePlane([1,3,7], 10)
sage: p.d_neighbors((0,0,0))
[(-1, -1, 1), (-1, 0, 1), (-1, 1, 0), (-1, 1, 1), (0, -1, 1),
(0, 0, 0), (0, 0, 1), (0, 1, 0), (1, -1, 1), (1, 0, 0), (1, 0,
1), (1, 1, 0)]
```

dimension()

Returns the dimension of the ambiant space.

OUTPUT:

integer

EXAMPLES:

```
sage: d = DiscreteSubset(3)
sage: d.dimension()
3

sage: p = DiscreteBox([0,3], [0,3], [0,3], [0,3])
sage: p.dimension()
4
```

edges_iterator(*roots=None*)

Returns an iterator over the pair of points in self that are adjacents, i.e. their difference is a canonical vector.

It considers only points that are connected to the given roots.

INPUT:

- **roots** - list of some elements in self

EXAMPLES:

```
sage: p = DiscretePlane([1,pi,7], 1+pi+7, mu=0)
sage: root = vector((0,0,0))
sage: root.set_immutable()
sage: it = p.edges_iterator(roots=[root])
sage: next(it)
((0, 0, 0), (1, 0, 0))
sage: next(it)
((0, 0, 0), (0, 1, 0))
sage: next(it)
((0, 0, 0), (0, 0, 1))
sage: next(it)
((-1, 1, 0), (0, 1, 0))
sage: next(it)
((-2, 1, 0), (-1, 1, 0))
```

has_edge(*p*, *s*)

Returns whether it has the edge (p, s) where s-p is a canonical vector.

INPUT:

- p - point in the space
- s - point in the space

EXAMPLES:

```
sage: F = lambda p: Integers(7)(2*p[0]+5*p[1])
sage: edge_predicate = lambda p,s: F(p) < F(s)
sage: D = DiscreteSubset(dimension=3, edge_predicate=edge_predicate)
sage: D.has_edge(vector((0,0)),vector((1,0)))
True
sage: D.has_edge(vector((0,0)),vector((-1,0)))
True
sage: D.has_edge(vector((-1,1)),vector((1,0)))
False
```

level_iterator(*roots=None*)

INPUT:

- roots - iterator of some elements in self

EXAMPLES:

```
sage: p = DiscreteSubset(3)
sage: root = vector((0,0,0))
sage: root.set_immutable()
sage: it = p.level_iterator(roots=[root])
sage: sorted(next(it))
[(0, 0, 0)]
sage: sorted(next(it))
[(-1, 0, 0), (0, -1, 0), (0, 0, -1), (0, 0, 1), (0, 1, 0), (1, 0, 0)]
sage: sorted(next(it))
[(-2, 0, 0),
 (-1, -1, 0),
 (-1, 0, -1),
 (-1, 0, 1),
 (-1, 1, 0),
 (0, -2, 0),
 (0, -1, -1),
 (0, -1, 1),
 (0, 0, -2),
 (0, 0, 2),
 (0, 1, -1),
 (0, 1, 1),
 (0, 2, 0),
 (1, -1, 0),
 (1, 0, -1),
 (1, 0, 1),
 (1, 1, 0),
 (2, 0, 0)]

sage: p = DiscretePlane([1,pi,7], 1+pi+7, mu=0)
sage: root = vector((0,0,0))
sage: root.set_immutable()
sage: it = p.level_iterator(roots=[root])
sage: sorted(next(it))
[(0, 0, 0)]
sage: sorted(next(it))
[(0, 0, 1), (0, 1, 0), (1, 0, 0)]
sage: sorted(next(it))
[(-1, 0, 1),
 (-1, 1, 0),
 (0, -1, 1),
 (0, 1, 1),
 (0, 2, 0),
 (1, 0, 1),
```

```
 (1, 1, 0),
 (2, 0, 0)]
```

list()

Return the list of elements in self.

EXAMPLES:

```
sage: P = DiscretePlane([3,4,5], 12, mu=20)
sage: tube = DiscreteTube([0,2],[0,2])
sage: I = P & tube
sage: sorted(I.list())
[(-3, -1, -1), (-3, -1, 0), (-2, -2, -1), (-2, -2, 0), (-2, -1,
-1), (-2, -1, 0), (-2, 0, -1), (-1, -1, -1)]
```

plot(*frame=False*, *edgecolor='blue'*, *pointcolor='blue'*)

Return a plot (2d or 3d) of the points and edges of self.

INPUT:

- `frame` - (default: False) if True, draw a bounding frame with labels
- `edgecolor` – string (default: `'blue'`), the color of the edges
- `pointcolor` – string (default: `'blue'`), the color of the points

EXAMPLES:

2d example:

```
sage: box = DiscreteBox([-5,5],[-5,5])
sage: box.plot() # optional long
```

3d example:

```
sage: P = DiscretePlane([1,3,7], 11)
sage: tube = DiscreteTube([-5,5],[-5,5])
sage: I = P & tube
sage: I.plot() # optional long
```

plot_cubes(***kwds*)

Returns the discrete object as cubes in 3d.

EXAMPLES:

```
sage: P = DiscretePlane([3,4,5], 12, mu=20)
sage: tube = DiscreteTube([-5,5],[-5,5])
sage: I = P & tube
sage: I.plot_cubes(color='red', frame_thickness=1 # optional long)
```

TESTS:

```
sage: box = DiscreteBox([-5,5],[-5,5])
sage: box.plot_cubes() # optional long
Traceback (most recent call last):
...
ValueError: this method is currently implemented only for objects living in 3 dimensions
```

plot_edges(*roots=None*, *color='blue'*, *m=None*)

Returns the mesh of the plane. The mesh is the union of segments joining two adjacents points.

INPUT:

- `roots` - list of some elements in self
- `color` – string (default: `'blue'`), the color of the edges

- m – projection matrix (default: None), it can be one of the following:
  - None - no projection is done
  - 'isometric' - the isometric projection
  - matrix - a 2 x 3 matrix
  - 'belle' - shortcut for matrix(2, [1/3.0, 1, 0, 2/3.0, 0, 1])
  - vector - defines the projection on the plane orthogonal to the vector.

EXAMPLES:

A 2d plot of a 2d object:

```
sage: D = DiscreteSubset(2)
sage: box = DiscreteBox([-5,5],[-5,5])
sage: I = D & box
sage: I.plot_edges(color='green') # optional long
```

A 3d plot of a 3d object:

```
sage: D = DiscreteSubset(3)
sage: box = DiscreteBox([-3,3],[-3,3],[-3,3])
sage: I = D & box
sage: I.plot_edges(color='green') # optional long
```

A 2d plot of a 3d object:

```
sage: D = DiscreteSubset(3)
sage: box = DiscreteBox([-3,3],[-3,3],[-3,3])
sage: I = D & box
sage: I.plot_edges(color='green', m='isometric') # optional long
```

plot_points(*color='blue'*, *m=None*)

Returns a 2d or 3d graphics object of the points of self.

INPUT:

- color – string (default: 'blue'), the color of the points
- m – projection matrix (default: None), it can be one of the following:
  - None - no projection is done
  - 'isometric' - the isometric projection
  - matrix - a 2 x n projection matrix
  - 'belle' - shortcut for matrix(2, [1/3.0, 1, 0, 2/3.0, 0, 1])
  - vector - defines the projection on the plane orthogonal to the vector.

EXAMPLES:

A 2d plot of a 2d object:

```
sage: D = DiscreteSubset(2)
sage: box = DiscreteBox([-5,5],[-5,5])
sage: I = D & box
sage: I.plot_points(color='green')      # optional long
```

A 3d plot of a 3d object:

```
sage: D = DiscreteSubset(3)
sage: box = DiscreteBox([-5,5],[-5,5],[-5,5])
sage: I = D & box
sage: I.plot_points(color='green')      # optional long
```

A 2d plot of a 3d object:

```
sage: D = DiscreteSubset(3)
sage: box = DiscreteBox([-5,5],[-5,5],[-5,5])
sage: I = D & box
sage: I.plot_points(color='green', m='isometric')      # optional long
```

plot_points_at_distance(*k*, *roots=None*, *color='blue'*, *projmat=None*)

Plot points at distance k from the roots.

INPUT:

- **k** - integer

EXAMPLES:

```
sage: alpha = solve(x+x**2+x**3==1, x)[2].right()
sage: vv = vector((alpha, alpha+alpha**2, 1))
sage: omega = (1+alpha)**2 / 2
sage: Pr = DiscretePlane(vv, omega, mu=pi, prec=200)
sage: Pr.plot_points_at_distance(200)               # optional long
sage: Pr.plot_points_at_distance(200, projmat='isometric') # optional long
```

projection_matrix(*m='isometric'*, *oblique=None*)

Return a projection matrix.

INPUT:

- **m** – projection matrix (default: `'isometric'`), it can be one of the following:
  - `'isometric'` - the isometric projection is used by default
  - matrix - a 2 x 3 matrix
  - `'belle'` - shortcut for `matrix(2, [1/3.0, 1, 0, 2/3.0, 0, 1])`
  - vector - defines the projection on the plane orthogonal to the vector.
- **oblique** – vector (default: `None`), vector perpendicular to the range space

EXAMPLES:

```
sage: d = DiscreteSubset(3)
sage: d.projection_matrix(vector((2,3,4)))
[  1.00000000000000  0.000000000000000 -0.500000000000000]
[ 0.000000000000000   1.00000000000000 -0.750000000000000]
sage: d.projection_matrix((2,3,4))
[  1.00000000000000  0.000000000000000 -0.500000000000000]
[ 0.000000000000000   1.00000000000000 -0.750000000000000]
sage: d.projection_matrix()
[-0.866025403784  0.866025403784             0.0]
[           -0.5            -0.5             1.0]
sage: d.projection_matrix(_)
[-0.866025403784439  0.866025403784439  0.000000000000000]
[-0.500000000000000 -0.500000000000000   1.00000000000000]
sage: d.projection_matrix('belle')
[0.333333333333            1.0            0.0]
[0.666666666667            0.0            1.0]
```

**DiscreteSubset.tikz(projmat=[-0.866025403784439 0.866025403784439 0.000000000000000]
[-0.500000000000000 -0.500000000000000   1.00000000000000], scale=1, clip=[], contour=[], edges=Tru**

INPUT:

- **projmat** – (default: M3to2) 2 x dim projection matrix where dim is the dimensoin of self, the isometric projection is used by default

- `scale` – real number (default: 1), scaling constant for the whole figure
- `clip` - list (default: `[]`), list of points whose convex hull describes a cliping path
- `contour` - list (default: `[]`), list of points describing a contour path to be drawn
- `edges` - bool (default: `True`), whether to draw edges
- `points` - bool (default: `True`), whether to draw points
- `axes` - bool (default: `False`), whether to draw axes
- `point_kwds` - dict (default: `{}`)
- `edge_kwds` - dict (default: `{}`)
- `axes_kwds` - dict (default: `{}`)
- `extra_code` – string (default: `''`), extra tikz code to add

EXAMPLES:

```
sage: p = DiscretePlane([2,3,5], 10)
sage: p.tikz(points=False, edges=False)
\begin{tikzpicture}
[scale=1]
\end{tikzpicture}
```

tikz_axes(*xshift=0*, *yshift=0*, *label='e'*, *projmat='isometric'*)

Return the tikz code for drawing axes.

INPUT:

- `xshift` - integer (default: `0`), x shift
- `yshift` - integer (default: `0`), y shift
- `label` - string (default: `"e"`), label for base vectors
- `projmat` - matrix (default: `'isometric'`), projection matrix

OUTPUT:

string

EXAMPLES:

2d example:

```
sage: d = DiscreteSubset(2)
sage: d.tikz_axes()
%the axes
\begin{scope}[xshift=0cm,yshift=0cm]
\draw[->,>=latex, very thick, blue] (0,0) -- (1, 0);
\draw[->,>=latex, very thick, blue] (0,0) -- (0, 1);
\node at (1.40000000000000,0) {$e_1$};
\node at (0,1.40000000000000) {$e_2$};
\end{scope}
```

3d example:

```
sage: d = DiscreteSubset(3)
sage: d.tikz_axes(projmat='isometric')
%the axes
\begin{scope}
[x={(-0.866025cm,-0.500000cm)}, y={(0.866025cm,-0.500000cm)},
z={(0.000000cm,1.000000cm)}, scale=1,xshift=0,yshift=0]
\draw[fill=white] (2,0,0) rectangle (-1.8,.1,1);
\draw[->,>=latex, very thick, blue] (0,0,0) -- (1, 0, 0);
\draw[->,>=latex, very thick, blue] (0,0,0) -- (0, 1, 0);
```

```
\draw[->,>=latex, very thick, blue] (0,0,0) -- (0, 0, 1);
\node at (1.40000000000000,0,0) {$e_1$};
\node at (0,1.40000000000000,0) {$e_2$};
\node at (0,0,1.40000000000000) {$e_3$};
\end{scope}
```

tikz_edges(*style='very thick'*, *color='blue'*, *roots=None*, *projmat=None*)

Returns the mesh of the object. The mesh is the union of segments joining two adjacents points.

INPUT:

- `style` - string (default: `'dashed, very thick'`)
- `color` - string or callable (default: `'blue'`), the color of all edges or a function : (u,v) -> color of the edge (u,v)
- `roots` - list of some elements in self
- `projmat` - matrix (default: `None`), projection matrix, if None, no projection is done.

EXAMPLES:

```
sage: p = DiscretePlane([2,3,5], 4)
sage: p.tikz_edges()
\draw[very thick, blue] (0, 0, 0) -- (1, 0, 0);
\draw[very thick, blue] (0, 0, 0) -- (0, 1, 0);
\draw[very thick, blue] (-1, 1, 0) -- (0, 1, 0);

sage: p.tikz_edges(color='orange')
\draw[very thick, orange] (0, 0, 0) -- (1, 0, 0);
\draw[very thick, orange] (0, 0, 0) -- (0, 1, 0);
\draw[very thick, orange] (-1, 1, 0) -- (0, 1, 0);

sage: c = lambda u,v: 'red' if u == 0 else 'blue'
sage: p.tikz_edges(color=c)
\draw[very thick, red] (0, 0, 0) -- (1, 0, 0);
\draw[very thick, red] (0, 0, 0) -- (0, 1, 0);
\draw[very thick, blue] (-1, 1, 0) -- (0, 1, 0);

sage: from slabbe.discrete_subset import M3to2
sage: p.tikz_edges(projmat=M3to2)
\draw[very thick, blue] (0.00000, 0.00000) -- (-0.86603, -0.50000);
\draw[very thick, blue] (0.00000, 0.00000) -- (0.86603, -0.50000);
\draw[very thick, blue] (1.73205, 0.00000) -- (0.86603, -0.50000);
```

tikz_noprojection(*projmat=None*, *scale=1*, *clip=[]*, *edges=True*, *points=True*, *axes=False*, *point_kwds={}*, *edge_kwds={}*, *axes_kwds={}*, *extra_code=''*)

Return the tikz code of self.

In this version, the points are not projected. If the points are in 3d, the tikz 3d picture is used.

INPUT:

- `projmat` – (default: None) 2*3 projection matrix for drawing unit faces, the isometric projection is used by default
- `scale` – real number (default: 1), scaling constant for the whole figure
- `clip` - list (default: `[]`), list of points describing a cliping path once projected. Works only if `self.dimension()` is 2.
- `edges` - bool (default: `True`), whether to draw edges
- `points` - bool (default: `True`), whether to draw points
- `axes` - bool (default: `False`), whether to draw axes

- point_kwds - dict (default: {})
- edge_kwds - dict (default: {})
- axes_kwds - dict (default: {})
- extra_code – string (default: ''), extra tikz code to add

EXAMPLES:

Object in 2d:

```
sage: L = DiscreteLine([2,5], 2+5, mu=0)
sage: b = DiscreteBox([-5,5],[-5,5])
sage: I = L & b
sage: point_kwds = {'label':lambda p:2*p[0]+5*p[1],'label_pos':'above right'}
sage: tikz = I.tikz_noprojection(scale=0.5,point_kwds=point_kwds)
sage: len(tikz)
2659
sage: tikz
\begin{tikzpicture}
[scale=0.500000000000000]
\draw[very thick, blue] (0, 0) -- (1, 0);
\draw[very thick, blue] (0, 0) -- (0, 1);
...
\node[above right] at (-5, 2) {$0$};
\node[circle,fill=black,draw=black,minimum size=0.8mm,inner sep=0pt,] at (-5, 3) {};
\node[above right] at (-5, 3) {$5$};
\end{tikzpicture}
```

Object in 3d:

```
sage: p = DiscretePlane([1,3,7], 11)
sage: d = DiscreteTube([-5,5],[-5,5])
sage: I = p & d
sage: s = I.tikz_noprojection()
sage: lines = s.splitlines()
sage: len(lines)
321
sage: print '\n'.join(lines[:4])
\begin{tikzpicture}
[x={(-0.866025cm,-0.500000cm)}, y={(0.866025cm,-0.500000cm)},
z={(0.000000cm,1.000000cm)}, scale=1]
\draw[very thick, blue] (0, 0, 0) -- (1, 0, 0);
```

tikz_points(*size='0.8mm'*, *label=None*, *label_pos='right'*, *fill='black'*, *options=''*, *roots=None*, *filter=None*, *projmat=None*)

INPUT:

- size - string (default: '0.8mm'), size of the points
- label - function (default: None), print some label next to the point
- label_pos - function (default: 'right'), tikz label position
- fill - string (default: 'black'), fill color
- options - string (default: ''), author tikz node circle options
- roots - list of some elements in self
- filter - boolean function, if filter(p) is False, the point p is not drawn
- projmat - matrix (default: None), projection matrix, if None, no projection is done.

EXAMPLES:

```
sage: p = DiscreteBox([0,3], [0,3], [0,3])
sage: s = p.tikz_points()
```

```
sage: lines = s.splitlines()
sage: lines[0]
'\\node[circle,fill=black,draw=black,minimum size=0.8mm,inner sep=0pt,] at (0, 0, 0) {};'

sage: p = DiscretePlane([1,3,7], 11)
sage: d = DiscreteTube([-1,1],[-1,1])
sage: I = p & d
sage: I.tikz_points()
\node[circle,fill=black,draw=black,minimum size=0.8mm,inner sep=0pt,] at (0, 0, 0) {};
\node[circle,fill=black,draw=black,minimum size=0.8mm,inner sep=0pt,] at (1, 0, 0) {};
\node[circle,fill=black,draw=black,minimum size=0.8mm,inner sep=0pt,] at (0, 1, 0) {};
\node[circle,fill=black,draw=black,minimum size=0.8mm,inner sep=0pt,] at (0, 0, 1) {};
\node[circle,fill=black,draw=black,minimum size=0.8mm,inner sep=0pt,] at (0, 1, 1) {};
\node[circle,fill=black,draw=black,minimum size=0.8mm,inner sep=0pt,] at (1, 1, 0) {};
\node[circle,fill=black,draw=black,minimum size=0.8mm,inner sep=0pt,] at (1, 0, 1) {};
```

Using a filter on the points:

```
sage: I.tikz_points(filter=lambda x:sum(x)==1)
\node[circle,fill=black,draw=black,minimum size=0.8mm,inner sep=0pt,] at (1, 0, 0) {};
\node[circle,fill=black,draw=black,minimum size=0.8mm,inner sep=0pt,] at (0, 1, 0) {};
\node[circle,fill=black,draw=black,minimum size=0.8mm,inner sep=0pt,] at (0, 0, 1) {};
```

tikz_projection_scale(*projmat='isometric'*, *scale=1*, *extra=''*)

INPUT:

- `projmat` – (default: `'isometric'`) It can be one of the following:
  - `'isometric'` - the isometric projection is used by default
  - matrix - a 2 x 3 matrix
  - `'belle'` - shortcut for `matrix(2, [1/3.0, 1, 0, 2/3.0, 0, 1])`
  - vector - defines the projection on the plane orthogonal to the vector.
- `scale` – real number (default: 1), scaling constant for the whole figure
- `extra` – string (default: '')

EXAMPLES:

```
sage: p = DiscretePlane([1,3,7], 11)
sage: p.tikz_projection_scale()
[x={(-0.866025cm,-0.500000cm)}, y={(0.866025cm,-0.500000cm)},
z={(0.000000cm,1.000000cm)}, scale=1]
sage: p.tikz_projection_scale(extra="xshift=4cm")
[x={(-0.866025cm,-0.500000cm)}, y={(0.866025cm,-0.500000cm)},
z={(0.000000cm,1.000000cm)}, scale=1,xshift=4cm]
```

**DiscreteTube(projmat=[-0.866025403784439 0.866025403784439 0.000000000000000]
[-0.500000000000000 -0.500000000000000 1.00000000000000], *args, **kwds)**

Bases: slabbe.discrete_subset.DiscreteSubset

Discrete Tube (preimage of a box by a projection matrix)

Subset of a discrete object such that its projection by a matrix is inside a certain box.

INPUT:

- `*args` - intervals, lists of size two : [min, max]
- `projmat` - matrix (default: `M3to2`), projection matrix

EXAMPLES:

```
sage: DiscreteTube([-5,5],[-5,5])
DiscreteTube: Preimage of [-5, 5] x [-5, 5] by a 2 by 3 matrix

sage: m = matrix(3,4,range(12))
sage: DiscreteTube([2,10],[3,4],[6,7], projmat=m)
DiscreteTube: Preimage of [2, 10] x [3, 4] x [6, 7] by a 3 by 4 matrix
```

EXAMPLES:

```
sage: p = DiscretePlane([1,pi,7], 1+pi+7, mu=0)
sage: tube = DiscreteTube([-5,5],[-5,5])
sage: I = p & tube
sage: I
Intersection of the following objects:
Set of points x in ZZ^3 satisfying: 0 <= (1, pi, 7) . x + 0 < pi + 8
DiscreteTube: Preimage of [-5, 5] x [-5, 5] by a 2 by 3 matrix
sage: len(list(I))
115
```

DiscreteTube.**clip**(*space=1*)

Return a good clip rectangle for this box.

INPUT:

- space – number (default: 1), inner space within the box

EXAMPLES:

```
sage: tube = DiscreteTube([-6,6],[-4,3])
sage: tube.clip()
[(-5, -3), (5, -3), (5, 2), (-5, 2), (-5, -3)]
```

class slabbe.discrete_subset.**Intersection**(*objets*)

Bases: slabbe.discrete_subset.DiscreteSubset

Intersection

todo:

- Rendre l'heritage 3d automatique

INPUT:

- objets - un tuple d'objets discrets

EXAMPLES:

Intersection de deux plans:

```
sage: p = DiscretePlane([1,3,7],11)
sage: q = DiscretePlane([1,3,5],9)
sage: Intersection((p,q))
Intersection of the following objects:
Set of points x in ZZ^3 satisfying: 0 <= (1, 3, 7) . x + 0 < 11
Set of points x in ZZ^3 satisfying: 0 <= (1, 3, 5) . x + 0 < 9
```

Shortcut:

```
sage: p & q
Intersection of the following objects:
Set of points x in ZZ^3 satisfying: 0 <= (1, 3, 7) . x + 0 < 11
Set of points x in ZZ^3 satisfying: 0 <= (1, 3, 5) . x + 0 < 9
```

Intersection of a plane and a tube:

```
sage: p = DiscretePlane([1,pi,7], 1+pi+7, mu=0)
sage: d = DiscreteTube([-5,5],[-5,5])
sage: I = p & d
sage: I
Intersection of the following objects:
Set of points x in ZZ^3 satisfying: 0 <= (1, pi, 7) . x + 0 < pi + 8
DiscreteTube: Preimage of [-5, 5] x [-5, 5] by a 2 by 3 matrix
sage: len(list(I))
115
```

Intersection of a line and a box:

```
sage: L = DiscreteLine([2,5], 2+5, mu=0)
sage: b = DiscreteBox([-5,5],[-5,5])
sage: I = L & b
sage: I
Intersection of the following objects:
Set of points x in ZZ^2 satisfying: 0 <= (2, 5) . x + 0 < 7
[-5, 5] x [-5, 5]
```

TESTS:

Intersected objects must be of the same dimension:

```
sage: box = DiscreteBox([-5,5],[-5,5])
sage: p = DiscretePlane([1,pi,7], 1+pi+7)
sage: p & box
Traceback (most recent call last):
...
ValueError: Intersection not defined for objects not of the same dimension
```

an_element()

Returns an element in self.

EXAMPLES:

```
sage: P = DiscretePlane([4,6,7], 17, mu=0)
sage: tube = DiscreteTube([-6.4, 6.4], [-5.2, 5.2])
sage: I = tube & P
sage: I.an_element()
(0, 0, 0)
sage: I.an_element() in I
True
```

TESTS:

```
sage: P = DiscretePlane([4,6,7], 17, mu=0)
sage: def contain(p): return 0 < P._v.dot_product(p) + P._mu <= P._omega
sage: P._predicate = contain
sage: tube = DiscreteTube([-6.4, 6.4], [-5.2, 5.2])
sage: I = tube & P
sage: I.an_element()
(0, 0, 0) not in the plane
trying similar points
(0, 0, 1)
```

has_edge(*p*, *s*)

Returns whether it has the edge (p, s) where s-p is a canonical vector.

INPUT:

- **p** - point in the space
- **s** - point in the space

EXAMPLES:

```
sage: d3 = DiscreteSubset(3)
sage: p = DiscretePlane([1,pi,7], 1+pi+7, mu=0)
sage: I = p & d3
sage: I.has_edge(vector((0,0,0)),vector((0,0,1)))
True
sage: I.has_edge(vector((0,0,0)),vector((0,0,-1)))
False
```

TESTS:

```
sage: F = lambda p: (2*p[0]+5*p[1]) % 7
sage: edge_predicate = lambda p,s: F(p) < F(s)
sage: D = DiscreteSubset(dimension=2, edge_predicate=edge_predicate)
sage: b = DiscreteBox([-5,5],[-5,5])
sage: I = D & b
sage: all(I.has_edge(a,b) for a,b in I.edges_iterator())
True
sage: all(D.has_edge(a,b) for a,b in I.edges_iterator())
True

sage: C = ChristoffelGraph((2,5))
sage: b = DiscreteBox([-5,5],[-5,5])
sage: I = C & b
sage: all(I.has_edge(a,b) for a,b in I.edges_iterator())
True
sage: all(C.has_edge(a,b) for a,b in I.edges_iterator())
True
```

slabbe.discrete_subset.**convex_boundary**(*L*)

EXAMPLES:

```
sage: from slabbe.discrete_subset import convex_boundary
sage: convex_boundary([(3,4), (1,2), (3,5)])
[(3, 5), (1, 2), (3, 4)]
```

1.2 Discrete Plane

Intersection of a plane and a tube:

```
sage: p = DiscretePlane([1,pi,7], 1+pi+7, mu=0)
sage: d = DiscreteTube([-5,5],[-5,5])
sage: I = p & d
sage: I
Intersection of the following objects:
Set of points x in ZZ^3 satisfying: 0 <= (1, pi, 7) . x + 0 < pi + 8
DiscreteTube: Preimage of [-5, 5] x [-5, 5] by a 2 by 3 matrix
sage: len(list(I))
115
```

Intersection of a line and a box:

```
sage: L = DiscreteLine([pi,sqrt(2)], pi+sqrt(2), mu=0)
sage: b = DiscreteBox([-5,5],[-5,5])
sage: I = L & b
sage: I
Intersection of the following objects:
Set of points x in ZZ^2 satisfying: 0 <= (pi, sqrt(2)) . x + 0 < pi + sqrt(2)
[-5, 5] x [-5, 5]
```

TODO:

- do some dimension checking for DiscreteLine and DiscretePlane

class slabbe.discrete_plane.**DiscreteHyperplane**(*v*, *omega*, *mu=0*, *prec=None*)

Bases: slabbe.discrete_subset.DiscreteSubset

This is the set of point p such that

$$0 \leq p \cdot v - mu < \omega$$

INPUT:

- v - normal vector
- omega - width
- mu - intercept (optional, default: 0)

EXAMPLES:

```
sage: L = DiscreteLine([pi,sqrt(2)], pi+sqrt(2), mu=10)
sage: L
Set of points x in ZZ^2 satisfying: 0 <= (pi, sqrt(2)) . x + 10 < pi + sqrt(2)

sage: p = DiscretePlane([1,pi,7], 1+pi+7, mu=0)
sage: p
Set of points x in ZZ^3 satisfying: 0 <= (1, pi, 7) . x + 0 < pi + 8

sage: p = DiscreteHyperplane([1,3,7,9], 20, mu=13)
sage: p
Set of points x in ZZ^4 satisfying: 0 <= (1, 3, 7, 9) . x + 13 < 20
```

TESTS:

```
sage: p = DiscretePlane([1,pi,7], 1+pi+7, mu=20)
sage: vector((0,0,0)) in p
False
sage: p = DiscretePlane([1,pi,7], 1+pi+7, mu=0)
sage: vector((0,0,0)) in p
True

sage: p = DiscreteHyperplane((2,3,4,5), 10)
sage: p.dimension()
4

sage: L = DiscreteLine([1,pi], 1+pi, mu=20)
sage: vector((0,0)) in L
False
sage: L = DiscreteLine([1,pi], 1+pi, mu=0)
sage: vector((0,0)) in L
True
```

an_element(*x=0*, *y=0*)

Returns an element in self.

EXAMPLES:

```
sage: p = DiscreteHyperplane([1,pi,7], 1+pi+7, mu=10)
sage: p.an_element()
(0, 0, 0)

sage: L = DiscreteLine([pi,sqrt(2)], pi+sqrt(2), mu=10)
sage: L.an_element()
(-2, -2)

sage: L = DiscreteLine([pi,sqrt(2)], pi+sqrt(2), mu=0)
sage: L.an_element()
(0, 0)
```

level_value(*p*)
Return the level value of a point p.

INPUT:

- p - point in the space

EXAMPLES:

```
sage: H = DiscreteHyperplane([1,3,7,9], 20, mu=13)
sage: p = H._space((1,2,3,4))
sage: H.level_value(p)
64
```

slabbe.discrete_plane.**DiscreteLine**
alias of DiscreteHyperplane

slabbe.discrete_plane.**DiscretePlane**
alias of DiscreteHyperplane

1.3 Discrete Lines

EXAMPLES:

```
sage: b = BilliardCube((1,pi,sqrt(2)))
sage: b
Cubic billiard of direction (1, pi, sqrt(2))
```

TODO:

- Should handle any direction
- Should use Forest structure for enumeration
- Should use +e_i only for children
- Fix documentation of class
- Fix issue with the assertion error in the step iterator
- not robust for non integral start point

class slabbe.billiard.**BilliardCube**(*v*, *start=(0, 0, 0)*)
Bases: slabbe.discrete_subset.Intersection

This is the set of point p such that

$$0 \leq p \cdot v - mu < \omega \text{ #fix me } 0 \leq p \cdot v - mu < \omega \text{ #fix me } 0 \leq p \cdot v - mu < \omega \text{ #fix me}$$

INPUT:

- v - directive vector
- start - initial point (default = (0,0,0))

EXAMPLES:

```
sage: b = BilliardCube((1,pi,sqrt(2)))
sage: b
Cubic billiard of direction (1, pi, sqrt(2))
```

```
sage: b = BilliardCube((1,pi,sqrt(2)))
sage: it = iter(b)
sage: [next(it) for _ in range(20)]
```

```
[(0, 0, 0),
 (0, 1, 0),
 (0, 1, 1),
 (0, 2, 1),
 (1, 2, 1),
 (1, 3, 1),
 (1, 3, 2),
 (1, 4, 2),
 (1, 5, 2),
 (2, 5, 2),
 (2, 6, 2),
 (2, 6, 3),
 (2, 7, 3),
 (2, 8, 3),
 (2, 8, 4),
 (3, 8, 4),
 (3, 9, 4),
 (3, 10, 4),
 (3, 10, 5),
 (3, 11, 5)]
```

::

```
sage: b = BilliardCube((1,sqrt(2),pi), start=(11,13,14))
sage: b.to_word()
word: 32313233132332133231323312333213323132333...
```

an_element()

Returns an element in self.

EXAMPLES:

```
sage: b = BilliardCube((1,pi,sqrt(2)))
sage: b.an_element()
(0, 0, 0)
```

children(*p*)

Return the children of a point.

This method overwrites the methods slabbe.discrete_subset.DiscreteSubset.children(), because for billiard words, we go only in one direction is each axis.

EXAMPLES:

```
sage: p = DiscretePlane([1,pi,7], 1+pi+7, mu=0)
sage: list(p.children(vector((0,0,0))))
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

connected_component_iterator(*roots=None*)

Return an iterator over the connected component of the root.

This method overwrites the methods slabbe.discrete_subset.DiscreteSubset.connected_component_iterator(because for billiard words, we go only in one direction is each axis which allows to use a forest structure for the enumeration.

INPUT:

- roots - list of some elements immutable in self

EXAMPLES:

```
sage: p = BilliardCube([1,pi,sqrt(7)])
sage: root = vector((0,0,0))
sage: root.set_immutable()
sage: it = p.connected_component_iterator(roots=[root])
sage: [next(it) for _ in range(5)]
[(0, 0, 0), (0, 1, 0), (0, 1, 1), (0, 2, 1), (1, 2, 1)]

sage: p = BilliardCube([1,pi,7.45], start=(10.2,20.4,30.8))
sage: it = p.connected_component_iterator()
sage: [next(it) for _ in range(5)]
[(10.2000000000000, 20.4000000000000, 30.8000000000000),
 (10.2000000000000, 20.4000000000000, 31.8000000000000),
 (10.2000000000000, 21.4000000000000, 31.8000000000000),
 (10.2000000000000, 21.4000000000000, 32.8000000000000),
 (10.2000000000000, 21.4000000000000, 33.8000000000000)]
```

step_iterator()

Return an iterator coding the steps of the discrete line.

EXAMPLES:

```
sage: b = BilliardCube((1,pi,sqrt(2)))
sage: it = b.step_iterator()
sage: [next(it) for _ in range(5)]
[(0, 1, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (0, 1, 0)]
```

TESTS:

Fix this:

```
sage: B = BilliardCube((1.1,2.2,3.3))
sage: B.to_word()
Traceback (most recent call last):
...
AssertionError: step(=(-1, 0, 1)) is not a canonical basis
vector.
```

to_word(*alphabet=[1, 2, 3]*)

Return the billiard word.

INPUT:

- `alphabet` - list

EXAMPLES:

```
sage: b = BilliardCube((1,pi,sqrt(2)))
sage: b.to_word()
word: 2321232212322312232123221322231223212322...

sage: B = BilliardCube((sqrt(3),sqrt(5),sqrt(7)))
sage: B.to_word()
word: 3213213231232133213213231232132313231232...
```

1.4 Christoffel Graph

This module was developped for the article on a d-dimensional extension of Christoffel Words written with Christophe Reutenauer [LR2014].

EXAMPLES:

Christoffel graph in 2d (tikz code):

```
sage: C = ChristoffelGraph((2,5))
sage: b = DiscreteBox([-5,5],[-5,5])
sage: I = C & b
sage: point_kwds = {'label':lambda p:C.level_value(p),'label_pos':'above right'}
sage: tikz = I.tikz_noprojection(scale=0.8,point_kwds=point_kwds)
```

Christoffel graph in 3d (tikz code):

```
sage: C = ChristoffelGraph((2,3,5))
sage: tikz = C.tikz_kernel()
```

TODO:

- Clean kernel_vector method of ChristoffelGraph

class slabbe.christoffel_graph.**ChristoffelGraph**(*v*, *mod=None*)

Bases: slabbe.discrete_subset.DiscreteSubset

Subset of a discrete object such that its projection by a matrix is inside a certain box.

INPUT:

- **v** - vector, normal vector

EXAMPLES:

```
sage: ChristoffelGraph((2,5))
Christoffel set of edges for normal vector v=(2, 5)
```

```
sage: C = ChristoffelGraph((2,5))
sage: it = C.edges_iterator()
sage: it.next()
((0, 0), (1, 0))
```

```
sage: C = ChristoffelGraph((2,5,8))
sage: it = C.edges_iterator()
sage: it.next()
((0, 0, 0), (1, 0, 0))
```

```
sage: C = ChristoffelGraph((2,5))
sage: b = DiscreteBox([-5,5],[-5,5])
sage: I = C & b
sage: point_kwds = {'label':lambda p:C.level_value(p),'label_pos':'above right'}
sage: tikz = I.tikz_noprojection(scale=0.8,point_kwds=point_kwds)
```

TEST:

This was once a bug. We make sure it is fixed:

```
sage: C = ChristoffelGraph((2,3,5))
sage: isinstance(C, DiscreteSubset)
True
```

has_edge(*p*, *s*)

Returns whether it has the edge (p, s) where s-p is a canonical vector.

INPUT:

- **p** - point in the space
- **s** - point in the space

EXAMPLES:

```
sage: C = ChristoffelGraph((2,5,8))
sage: C.has_edge(vector((0,0,0)), vector((0,0,1)))
True
sage: C.has_edge(vector((0,0,0)), vector((0,0,2)))
False
sage: C.has_edge(vector((0,0,0)), vector((0,0,-1)))
False

sage: C = ChristoffelGraph((2,5))
sage: C.has_edge(vector((0,0)),vector((1,0)))
True
sage: C.has_edge(vector((0,0)),vector((-1,0)))
False
sage: C.has_edge(vector((-1,1)),vector((1,0)))
False
```

kernel_vector(*way='LLL'*, *verbose=False*)

todo: clean this

EXAMPLES:

```
sage: C = ChristoffelGraph((2,5,7))
sage: C.kernel_vector()
[(-1, -1, 1), (3, -4, 0)]
```

level_value(*p*)

Return the level value of a point p.

INPUT:

- p - point in the space

EXAMPLES:

```
sage: C = ChristoffelGraph((2,5,8))
sage: C.level_value(vector((2,3,4)))
6
sage: C.level_value(vector((1,1,1)))
0
```

**ChristoffelGraph.tikz_kernel(projmat=[-0.866025403784439 0.866025403784439 0.000000000000000]
[-0.500000000000000 -0.500000000000000 1.00000000000000], scale=1, edges=True, points=True, label**

INPUT:

- projmat – (default: M3to2) 2 x dim projection matrix where dim is the dimensoin of self, the isometric projection is used by default
- scale – real number (default: 1), scaling constant for the whole figure
- edges - bool (optional, default: True), whether to draw edges
- points - bool (optional, default: True), whether to draw points
- point_kwds - dict (default: {})
- edge_kwds - dict (default: {})
- extra_code – string (default: ''), extra tikz code to add
- way – string (default: 'LLL'), the way the base of the kernel is computed
- kernel_vector – list (default: None), the vectors, if None it uses kernel_vector() output.

EXAMPLES:

```
sage: C = ChristoffelGraph((2,3,5))
sage: tikz = C.tikz_kernel()
sage: lines = tikz.splitlines()
sage: len(lines)
306
```

1.5 Double Square Tiles

If a polyomino P tiles the plane by translation, then there exists a regular tiling of the plane by P [WVL1984], i.e., where the set of translations forms a lattice. Such a polyomino was called *exact* by Wijshoff and van Leeuven. There are two types of regular tiling of the plane : square and hexagonal. These are characterized by the Beauquier-Nivat condition [BN1991]. Deciding whether a polyomino is exact can be done efficiently from the boundary and in linear time for square tiling [BFP2009]. Brlek, Fédou, Provençal also remarked that there exist polyominoes leading to more than one regular tilings but conjectured that any polyomino produces at most two regular square tilings. This conjecture was proved in [BBL2012]. In [BBGL2011], two infinite families of *double square tiles* were provided, that is polyominoes having exactly two distinct regular square tilings of the plane, namely the Christoffel tiles and the Fibonacci tiles. Finally, in [BGL2012], it was shown that any double square tile can be constructed using two simple combinatorial rules: EXTEND and SWAP.

This module is about double square tiles. Notations are chosen according to [BGL2012]. It allows to construct, study and show double square tiles. Operations TRIM, SWAP and EXTEND are implemented. Double square tiles can be shown using Sage 2D Graphics objects or using tikz.

REFERENCES:

AUTHORS:

- Sébastien Labbé, 2008: initial version
- Alexandre Blondin Massé, 2008: initial version
- Sébastien Labbé, March 2013: rewrite for inclusion into Sage

EXAMPLES:

Double Square tile from the boundary word of a known double square:

```
sage: DoubleSquare(words.fibonacci_tile(2))
Double Square Tile
  w0 = 32303010   w4 = 10121232
  w1 = 30323      w5 = 12101
  w2 = 21232303   w6 = 03010121
  w3 = 23212      w7 = 01030
(|w0|, |w1|, |w2|, |w3|) = (8, 5, 8, 5)
(d0, d1, d2, d3)         = (10, 16, 10, 16)
(n0, n1, n2, n3)         = (0, 0, 0, 0)


sage: DoubleSquare(christoffel_tile(4,7))
Double Square Tile
  w0 = 03                              w4 = 21
  w1 = 0103010103010301010301030       w5 = 2321232321232123232123212
  w2 = 10103010                        w6 = 32321232
  w3 = 1                               w7 = 3
(|w0|, |w1|, |w2|, |w3|) = (2, 25, 8, 1)
(d0, d1, d2, d3)         = (26, 10, 26, 10)
(n0, n1, n2, n3)         = (0, 2, 0, 0)
```

Double Square tile from the lengths of the w_i:

```
sage: DoubleSquare((4,7,4,7))
Double Square Tile
  w0 = 3232      w4 = 1010
  w1 = 1212323   w5 = 3030101
  w2 = 2121      w6 = 0303
  w3 = 0101212   w7 = 2323030
(|w0|, |w1|, |w2|, |w3|) = (4, 7, 4, 7)
(d0, d1, d2, d3)         = (14, 8, 14, 8)
(n0, n1, n2, n3)         = (0, 0, 0, 0)
```

DoubleSquare tile from the words (w_0, w_1, w_2, w_3):

```
sage: DoubleSquare(([3,2], [3], [0,3], [0,1,0,3,0]))
Double Square Tile
  w0 = 32              w4 = 10
  w1 = 3               w5 = 1
  w2 = 03              w6 = 21
  w3 = 01030           w7 = 23212
(|w0|, |w1|, |w2|, |w3|) = (2, 1, 2, 5)
(d0, d1, d2, d3)         = (6, 4, 6, 4)
(n0, n1, n2, n3)         = (0, 0, 0, 1)
```

Reduction of a double square tile:

```
sage: D = DoubleSquare(christoffel_tile(4,7))
sage: D.reduction()
['TRIM_1', 'TRIM_1', 'TRIM_2', 'TRIM_1', 'TRIM_0', 'TRIM_2']
sage: D.apply_reduction()
Double Square Tile
  w0 =     w4 =
  w1 = 0   w5 = 2
  w2 =     w6 =
  w3 = 1   w7 = 3
(|w0|, |w1|, |w2|, |w3|) = (0, 1, 0, 1)
(d0, d1, d2, d3)         = (2, 0, 2, 0)
(n0, n1, n2, n3)         = (0, NaN, 0, NaN)
```

The intermediate steps of the reduction of a double square tile:

```
sage: E,op = D.reduce()
sage: E
Double Square Tile
  w0 = 03                 w4 = 21
  w1 = 010301010301030    w5 = 232123232123212
  w2 = 10103010           w6 = 32321232
  w3 = 1                  w7 = 3
(|w0|, |w1|, |w2|, |w3|) = (2, 15, 8, 1)
(d0, d1, d2, d3)         = (16, 10, 16, 10)
(n0, n1, n2, n3)         = (0, 1, 0, 0)
sage: op
'TRIM_1'


sage: D.reduce_ntimes(3)
Double Square Tile
  w0 = 03      w4 = 21
  w1 = 01030   w5 = 23212
  w2 = 10      w6 = 32
  w3 = 1       w7 = 3
(|w0|, |w1|, |w2|, |w3|) = (2, 5, 2, 1)
(d0, d1, d2, d3)         = (6, 4, 6, 4)
(n0, n1, n2, n3)         = (0, 1, 0, 0)
```

Plot a double square tile and plot its reduction:

```
sage: D = DoubleSquare((34,21,34,21))
sage: D.plot()                # long time (1s)
sage: D.plot_reduction()      # long time (1s)
```

It is not said clear enough in the articles, but double square reduction also works for double square tiles that are 8-connected polyominoes:

```
sage: D = DoubleSquare((55,34,55,34))
sage: D.plot()                # long time (1s)
sage: D.plot_reduction()      # long time (1s)
```

class slabbe.double_square_tile.**DoubleSquare**(*data*, *rot180=None*, *steps=None*)

Bases: sage.structure.sage_object.SageObject

A double square tile.

We represent a double square tile by its boundary, that is a finite sequence on the alphabet $A = \{0, 1, 2, 3\}$ where 0 is a East step, 1 is a North step, 2 is a West step and 3 is a South step.

INPUT:

- `data` - can be one of the following:
  - word - word over over the alphabet A representing the boundary of a double square tile
  - tuple - tuple of 4 elements (w0,w1,w2,w3) or 8 elements (w0,w1,w2,w3,w4,w5,w6,w7) such that each wi is a sequence over the alphabet A. The condition $w_i w_{i+1} = hat(w_{i+4} w_{i+5})$ must be verified for all i modulo 8.
  - tuple - tuple of 4 integers, the lengths of (w0,w1,w2,w3)
- `rot180` - WordMorphism (default: None), involution on the alphabet A and representing a rotation of 180 degrees. If None, the morphism 0->2, 1->3, 2->0, 3->1 is considered.
- `steps` - dict (default: None), mapping letters of A to steps in the plane. If None, the corresondance 0->(1,0), 1->(0,1), 2->(-1,0), 3->(0,-1) is considered.

EXAMPLES:

From a double square:

```
sage: DoubleSquare(words.fibonacci_tile(1))
Double Square Tile
  w0 = 32   w4 = 10
  w1 = 3    w5 = 1
  w2 = 03   w6 = 21
  w3 = 0    w7 = 2
(|w0|, |w1|, |w2|, |w3|) = (2, 1, 2, 1)
(d0, d1, d2, d3)         = (2, 4, 2, 4)
(n0, n1, n2, n3)         = (1, 0, 1, 0)
sage: DoubleSquare(words.fibonacci_tile(2))
Double Square Tile
  w0 = 32303010   w4 = 10121232
  w1 = 30323      w5 = 12101
  w2 = 21232303   w6 = 03010121
  w3 = 23212      w7 = 01030
(|w0|, |w1|, |w2|, |w3|) = (8, 5, 8, 5)
(d0, d1, d2, d3)         = (10, 16, 10, 16)
(n0, n1, n2, n3)         = (0, 0, 0, 0)

sage: DoubleSquare(christoffel_tile(9,7))
Double Square Tile
  w0 = 03                          w4 = 21
  w1 = 0101030101030101030         w5 = 23232123232123232
  w2 = 10101030101030101030101010  w6 = 32323212323212323232123232
  w3 = 1                           w7 = 3
```

```
(|w0|, |w1|, |w2|, |w3|) = (2, 19, 24, 1)
(d0, d1, d2, d3)         = (20, 26, 20, 26)
(n0, n1, n2, n3)         = (0, 0, 1, 0)
```

From the w_i:

```
sage: D = DoubleSquare(([],[],[0,1,0,1],[0,1]))
sage: D.rot180
WordMorphism: 0->2, 1->3, 2->0, 3->1
sage: D._steps
{0: (1, 0), 1: (0, 1), 2: (-1, 0), 3: (0, -1)}
sage: D
Double Square Tile
  w0 =         w4 =
  w1 =         w5 =
  w2 = 0101    w6 = 3232
  w3 = 01      w7 = 32
(|w0|, |w1|, |w2|, |w3|) = (0, 0, 4, 2)
(d0, d1, d2, d3)         = (2, 4, 2, 4)
(n0, n1, n2, n3)         = (0, 0, 2, 0)
```

One may also provide strings as long as other arguments are consistent:

```
sage: steps = {'0':(1,0), '1':(0,1), '2':(-1,0), '3': (0,-1)}
sage: rot180 = WordMorphism('0->2,2->0,3->1,1->3')
sage: DoubleSquare(('','','0101','01','','','3232','32'), rot180, steps)
Double Square Tile
  w0 =         w4 =
  w1 =         w5 =
  w2 = 0101    w6 = 3232
  w3 = 01      w7 = 32
(|w0|, |w1|, |w2|, |w3|) = (0, 0, 4, 2)
(d0, d1, d2, d3)         = (2, 4, 2, 4)
(n0, n1, n2, n3)         = (0, 0, 2, 0)
```

The first four words wi are sufficient:

```
sage: steps = {'0':(1,0), '1':(0,1), '2':(-1,0), '3': (0,-1)}
sage: rot180 = WordMorphism('0->2,2->0,3->1,1->3')
sage: DoubleSquare(('','','0101','01'), rot180, steps)
Double Square Tile
  w0 =         w4 =
  w1 =         w5 =
  w2 = 0101    w6 = 3232
  w3 = 01      w7 = 32
(|w0|, |w1|, |w2|, |w3|) = (0, 0, 4, 2)
(d0, d1, d2, d3)         = (2, 4, 2, 4)
(n0, n1, n2, n3)         = (0, 0, 2, 0)
```

alphabet()

Returns the python set of the letters that occurs in the boundary word.

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D.alphabet()
set([0, 1, 2, 3])
```

apply(*L*)

Return the double square obtained after the application of a list of operations.

INPUT:

- L - list, list of strings

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D.apply(['SWAP_0', 'EXTEND_3', 'TRIM_3'])
Double Square Tile
  w0 = 01030323      w4 = 23212101
  w1 = 21232303010   w5 = 03010121232
  w2 = 30323212      w6 = 12101030
  w3 = 10121232303   w7 = 32303010121
(|w0|, |w1|, |w2|, |w3|) = (8, 11, 8, 11)
(d0, d1, d2, d3)         = (22, 16, 22, 16)
(n0, n1, n2, n3)         = (0, 0, 0, 0)

sage: D.apply(D.reduction())
Double Square Tile
  w0 =     w4 =
  w1 = 3   w5 = 1
  w2 =     w6 =
  w3 = 2   w7 = 0
(|w0|, |w1|, |w2|, |w3|) = (0, 1, 0, 1)
(d0, d1, d2, d3)         = (2, 0, 2, 0)
(n0, n1, n2, n3)         = (0, NaN, 0, NaN)
```

apply_morphism(*m*)

INPUT:

- m - a WordMorphism

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(1))
sage: m = WordMorphism({0:[0],1:[1,0,1],2:[2],3:[3,2,3]})
sage: D.apply_morphism(m)
Double Square Tile
  w0 = 3232   w4 = 1010
  w1 = 323    w5 = 101
  w2 = 0323   w6 = 2101
  w3 = 0      w7 = 2
(|w0|, |w1|, |w2|, |w3|) = (4, 3, 4, 1)
(d0, d1, d2, d3)         = (4, 8, 4, 8)
(n0, n1, n2, n3)         = (1, 0, 1, 0)
```

apply_reduction()

Apply the reduction algorithm on self.

This is equivalent to `self.apply(self.reduction())`.

EXAMPLES:

```
sage: D = DoubleSquare(christoffel_tile(9,7))
sage: D.apply_reduction()
Double Square Tile
  w0 =     w4 =
  w1 = 0   w5 = 2
  w2 =     w6 =
  w3 = 1   w7 = 3
(|w0|, |w1|, |w2|, |w3|) = (0, 1, 0, 1)
(d0, d1, d2, d3)         = (2, 0, 2, 0)
(n0, n1, n2, n3)         = (0, NaN, 0, NaN)

sage: D = DoubleSquare((5,7,4,13))
sage: D.apply_reduction()
Double Square Tile
  w0 =     w4 =
  w1 =     w5 =
  w2 = 1   w6 = 0
  w3 =     w7 =
(|w0|, |w1|, |w2|, |w3|) = (0, 0, 1, 0)
```

```
(d0, d1, d2, d3)         = (0, 1, 0, 1)
(n0, n1, n2, n3)         = (NaN, 0, NaN, 0)

sage: D = DoubleSquare((5,2,4,13))
sage: D.reduce_ntimes(3)
Double Square Tile
  w0 = 0    w4 = 0
  w1 = 12   w5 = 32
  w2 = 01   w6 = 03
  w3 = 2    w7 = 2
(|w0|, |w1|, |w2|, |w3|) = (1, 2, 2, 1)
(d0, d1, d2, d3)         = (3, 3, 3, 3)
(n0, n1, n2, n3)         = (0, 0, 0, 0)
sage: D.apply_reduction()
Traceback (most recent call last):
...
ValueError: not reducible, because self is nondegenerate and
d_0 == d_1 == 3. Also, the turning number (=0) must be +1 or -1
for the reduction to apply.
```

boundary_word()

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D.boundary_word()
Path: 32303010303232123230323212101212321210 10...
```

d(i)

Return the integer d_i.

The value of d_i is defined as $d_i = |w_{i-1}| + |w_{i+1}|$.

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: [D.d(i) for i in range(8)]
[10, 16, 10, 16, 10, 16, 10, 16]
```

extend(i)

Apply $EXTEND_i$ on self.

This adds a period of length d_i to w_i and w_{i+4}.

INPUT:

- **i** - integer

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D.extend(3)
Double Square Tile
  w0 = 32303010                 w4 = 10121232
  w1 = 30323                    w5 = 12101
  w2 = 21232303                 w6 = 03010121
  w3 = 2321210121230323212      w7 = 0103032303010121010 30
(|w0|, |w1|, |w2|, |w3|) = (8, 5, 8, 21)
(d0, d1, d2, d3)         = (26, 16, 26, 16)
(n0, n1, n2, n3)         = (0, 0, 0, 1)
```

factorization_points()

Returns the eight factorization points of this configuration

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(1))
sage: D.factorization_points()
[0, 2, 3, 5, 6, 8, 9, 11]
```

hat()

Return the hat function returning the reversal of a word path.

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D.w(0)
Path: 32303010
sage: D.hat(D.w(0))
Path: 23212101
```

height()

Returns the width of this polyomino, i.e. the difference between its uppermost and lowermost coordinates

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D.height()
9

sage: D = DoubleSquare((34,21,34,21))
sage: D.height()
23
```

is_degenerate()

Return whether self is degenerate.

A double square is *degenerate* if one of the w_i is empty.

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(1))
sage: D.is_degenerate()
False

sage: rot180 = WordMorphism('0->2,2->0,3->1,1->3')
sage: D = DoubleSquare(('','0','10','1'), rot180)
sage: D.is_degenerate()
True
```

is_flat()

Return whether self is flat.

A double square is *flat* if one of the $w_i w_{i+1}$ is empty.

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(1))
sage: D.is_flat()
False

sage: rot180 = WordMorphism('0->2,2->0,3->1,1->3')
sage: D = DoubleSquare(('','','0101','01'), rot180)
sage: D.is_flat()
True
```

is_morphic_pentamino()

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(1))
sage: D.is_morphic_pentamino()
True
```

is_singular()

Return whether self is singular.

A double square is *singular* if there exists i such that w_{i-1} and w_{i+1} are empty, equivalently if $d_i = 0$.

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(1))
sage: D.is_singular()
False

sage: rot180 = WordMorphism('0->2,2->0,3->1,1->3')
sage: D = DoubleSquare(('','03010','','1011'), rot180)
sage: D.is_singular()
True
```

latex_8_tuple()

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D.latex_8_tuple()
('{\\bf 32303010}', '{\\bf 30323}', '{\\bf 21232303}', '{\\bf 23212}',
 '{\\bf 10121232}', '{\\bf 12101}', '{\\bf 03010121}', '{\\bf 01030}')
```

latex_array()

Return a LaTeX array of self.

This code was used to create Table 1 in [BGL2012].

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(1))
sage: print D.latex_array()
\begin{array}{lllllll}
i & w_i & u_i & v_i & |w_i| & d_i & n_i
\\
\hline
0 & {\bf 32} & {\bf } & {\bf 32} & 2 & 2 & 1\\
1 & {\bf 3} & {\bf 3} & {\bf 032} & 1 & 4 & 0\\
2 & {\bf 03} & {\bf } & {\bf 03} & 2 & 2 & 1\\
3 & {\bf 0} & {\bf 0} & {\bf 103} & 1 & 4 & 0\\
4 & {\bf 10} & {\bf } & {\bf 10} & 2 & 2 & 1\\
5 & {\bf 1} & {\bf 1} & {\bf 210} & 1 & 4 & 0\\
6 & {\bf 21} & {\bf } & {\bf 21} & 2 & 2 & 1\\
7 & {\bf 2} & {\bf 2} & {\bf 321} & 1 & 4 & 0\\
\hline
\end{array}
```

latex_table()

Returns a Latex expression of a table containing the parameters of this double square.

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D.latex_table()
\begin{tabular}{|c|}
\hline
\\
\begin{tikzpicture}
[first/.style={circle,draw=black,fill=gray, inner sep=0pt, minimum size=3pt},
second/.style={rectangle,draw=black,fill=white, inner sep=0pt, minimum size=3pt}]
...
\end{tikzpicture} \\[1ex]
\hline\\
$(w_0,w_1,w_2,w_3) = (8,5,8,5)$ \\
```

```
$u_0 = 32303010$\quad $u_1 = 30323$$u_2 = 21232303$\quad $u_3 = 23212$\\
$v_0 = 30$\quad $v_1 = 21232303010$$v_2 = 23$\quad $v_3 = 10121232303$\\
$(n_0,n_1,n_2,n_3) = (0,0,0,0)$ \\
Turning number = -1\\
Self-avoiding = True\\
\hline
\end{tabular}
```

n(*i*)

Return the integer n_i.

The value of n_i is defined as the quotient of $|w_i|$ by d_i.

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: [D.n(i) for i in range(8)]
[0, 0, 0, 0, 0, 0, 0, 0]

sage: A = D.extend(1).extend(1).extend(1).extend(1)
sage: [A.n(i) for i in range(8)]
[0, 4, 0, 0, 0, 4, 0, 0]
```

If $d_i = 0$ then n_i is not defined:

```
sage: D = DoubleSquare(words.fibonacci_tile(1))
sage: B = D.reduce_ntimes(2)
sage: [B.n(i) for i in range(8)]
[0, NaN, 0, NaN, 0, NaN, 0, NaN]
```

plot(*pathoptions={'rgbcolor': 'black', 'thickness': 3}, fill=True, filloptions={'alpha': 0.2, 'rgbcolor': 'black'}, startpoint=True, startoptions={'pointsize': 100, 'rgbcolor': 'black'}, endarrow=True, arrowoptions={'width': 3, 'rgbcolor': 'black', 'arrowsize': 5}, gridlines=False, gridoptions={}, axes=False*)

Returns a 2d Graphics illustrating the double square tile associated to this configuration including the factorizations points.

INPUT:

- `pathoptions` - (dict, default:dict(rgbcolor='red',thickness=3)), options for the path drawing
- `fill` - (boolean, default: True), if fill is True and if the path is closed, the inside is colored
- `filloptions` - (dict, default:dict(rgbcolor='red',alpha=0.2)), ptions for the inside filling
- `startpoint` - (boolean, default: True), draw the start point?
- `startoptions` - (dict, default:dict(rgbcolor='red',pointsize=100)) options for the start point drawing
- `endarrow` - (boolean, default: True), draw an arrow end at the end?
- `arrowoptions` - (dict, default:dict(rgbcolor='red',arrowsize=20, width=3)) options for the end point arrow
- `gridlines`- (boolean, default: False), show gridlines?
- `gridoptions` - (dict, default: {}), options for the gridlines
- `axes` - (boolean, default: False), options for the axes

EXAMPLES:

The cross of area 5 together with its double square factorization points:

```
sage: D = DoubleSquare(words.fibonacci_tile(1))
sage: D.plot()              # long time (1s)
```

plot_reduction(*ncols=3*, *options={}*)

Return a graphics array of the reduction.

INPUT:

- `ncols` - integer (default: `3`), number of columns
- `options` - dict (default: `{}`), options given to the plot method of each double square

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D.plot_reduction()          # long time (1s)
```

Using the color options:

```
sage: p = dict(rgbcolor='red', thickness=1)
sage: q = dict(rgbcolor='blue', alpha=1)
sage: options = dict(endarrow=False,startpoint=False,pathoptions=p,filloptions=q)
sage: D.plot_reduction(options=options)      # long time (1s)
```

reduce()

Reduces self by the application of TRIM or otherwise SWAP.

INPUT:

- `self` - non singular double square tile on the alphabet $0, 1, 2, 3$ such that its turning number is +1 or -1.

OUTPUT:

- DoubleSquare - the reduced double square
- string - the operation which was performed

EXAMPLES:

```
sage: D = DoubleSquare((34,21,34,21))
sage: E,op = D.reduce()
sage: E
Double Square Tile
  w0 = 32303010                 w4 = 10121232
  w1 = 303232123230301030323    w5 = 1210103010121232121 01
  w2 = 21232303                 w6 = 03010121
  w3 = 2321210121232303232 12   w7 = 0103032303010121010 30
(|w0|, |w1|, |w2|, |w3|) = (8, 21, 8, 21)
(d0, d1, d2, d3)         = (42, 16, 42, 16)
(n0, n1, n2, n3)         = (0, 1, 0, 1)
sage: op
'SWAP_1'

sage: D = DoubleSquare((1,2,2,1))
sage: D
Double Square Tile
  w0 = 1    w4 = 1
  w1 = 23   w5 = 03
  w2 = 12   w6 = 10
  w3 = 3    w7 = 3
(|w0|, |w1|, |w2|, |w3|) = (1, 2, 2, 1)
(d0, d1, d2, d3)         = (3, 3, 3, 3)
(n0, n1, n2, n3)         = (0, 0, 0, 0)
sage: D.reduce()
Traceback (most recent call last):
...
ValueError: not reducible, because self is nondegenerate and
d_0 == d_1 == 3. Also, the turning number (=0) must be +1 or -1
for the reduction to apply.
```

TESTS:

```
sage: D = DoubleSquare((5,4,3,4))
sage: D
Double Square Tile
  w0 = 90128   w4 = 40123
  w1 = 7659    w5 = 7654
  w2 = 012     w6 = 012
  w3 = 3765    w7 = 8765
(|w0|, |w1|, |w2|, |w3|) = (5, 4, 3, 4)
(d0, d1, d2, d3)         = (8, 8, 8, 8)
(n0, n1, n2, n3)         = (0, 0, 0, 0)
sage: D.reduce()
Traceback (most recent call last):
...
ValueError: not reducible, because self is nondegenerate and
d_0 == d_1 == 8. Also, the turning number (=-1) must be +1 or
-1 for the reduction to apply.
```

reduce_ntimes(*iteration=1*)

Reduces the double square self until it is singular.

INPUT:

- `iteration` - integer (default: 1), number of iterations to perform

OUTPUT:

DoubleSquare

EXAMPLES:

```
sage: D = DoubleSquare((34,21,34,21))
sage: D.reduce_ntimes(10)
Double Square Tile
  w0 =     w4 =
  w1 = 3   w5 = 1
  w2 =     w6 =
  w3 = 2   w7 = 0
(|w0|, |w1|, |w2|, |w3|) = (0, 1, 0, 1)
(d0, d1, d2, d3)         = (2, 0, 2, 0)
(n0, n1, n2, n3)         = (0, NaN, 0, NaN)
```

reduction()

Return the list of operations to reduce self to a singular double square.

OUTPUT:

- list of strings

EXAMPLES:

```
sage: D = DoubleSquare((34,21,34,21))
sage: D.reduction()
['SWAP_1', 'TRIM_1', 'TRIM_3', 'SWAP_1', 'TRIM_1', 'TRIM_3', 'TRIM_0', 'TRIM_2']

sage: D = DoubleSquare(christoffel_tile(9,7))
sage: D.reduction()
['TRIM_2', 'TRIM_1', 'TRIM_1', 'TRIM_1', 'TRIM_0', 'TRIM_2', 'TRIM_2']
```

reverse()

Apply $REVERSE$ on self.

This reverses the words w_i.

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D
Double Square Tile
  w0 = 32303010   w4 = 10121232
  w1 = 30323      w5 = 12101
  w2 = 21232303   w6 = 03010121
  w3 = 23212      w7 = 01030
(|w0|, |w1|, |w2|, |w3|) = (8, 5, 8, 5)
(d0, d1, d2, d3)         = (10, 16, 10, 16)
(n0, n1, n2, n3)         = (0, 0, 0, 0)
sage: D.reverse()
Double Square Tile
  w0 = 21232      w4 = 03010
  w1 = 30323212   w5 = 12101030
  w2 = 32303      w6 = 10121
  w3 = 01030323   w7 = 23212101
(|w0|, |w1|, |w2|, |w3|) = (5, 8, 5, 8)
(d0, d1, d2, d3)         = (16, 10, 16, 10)
(n0, n1, n2, n3)         = (0, 0, 0, 0)
sage: D.reverse().reverse() == D
True
```

shift()

Apply $SHIFT$ on self.

This replaces w_i by w_{i+1}.

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D
Double Square Tile
  w0 = 32303010   w4 = 10121232
  w1 = 30323      w5 = 12101
  w2 = 21232303   w6 = 03010121
  w3 = 23212      w7 = 01030
(|w0|, |w1|, |w2|, |w3|) = (8, 5, 8, 5)
(d0, d1, d2, d3)         = (10, 16, 10, 16)
(n0, n1, n2, n3)         = (0, 0, 0, 0)
sage: D.shift()
Double Square Tile
  w0 = 30323      w4 = 12101
  w1 = 21232303   w5 = 03010121
  w2 = 23212      w6 = 01030
  w3 = 10121232   w7 = 32303010
(|w0|, |w1|, |w2|, |w3|) = (5, 8, 5, 8)
(d0, d1, d2, d3)         = (16, 10, 16, 10)
(n0, n1, n2, n3)         = (0, 0, 0, 0)
sage: D.shift().shift().shift().shift().shift().shift().shift().shift() == D
True
sage: D.shift().shift().shift().shift() == D
False
```

swap(i)

Apply $SWAP_i$ on self.

This replaces w_j by $w_{\hat{j+4}}$ for each $j = i, i+2, i+4, i+6$ and $w_j = (u_j * v_j)^{n_j} u_j$ by $(v_j * u_j)^{n_j} v_j$ for each $j = i+1, i+3, i+5, i+7$. This is an involution if the u_j are non empty.

INPUT:

- **i** - integer

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D
```

```
Double Square Tile
  w0 = 32303010   w4 = 10121232
  w1 = 30323      w5 = 12101
  w2 = 21232303   w6 = 03010121
  w3 = 23212      w7 = 01030
(|w0|, |w1|, |w2|, |w3|) = (8, 5, 8, 5)
(d0, d1, d2, d3)         = (10, 16, 10, 16)
(n0, n1, n2, n3)         = (0, 0, 0, 0)
sage: D.swap(1)
Double Square Tile
  w0 = 30      w4 = 12
  w1 = 32303   w5 = 10121
  w2 = 23      w6 = 01
  w3 = 21232   w7 = 03010
(|w0|, |w1|, |w2|, |w3|) = (2, 5, 2, 5)
(d0, d1, d2, d3)         = (10, 4, 10, 4)
(n0, n1, n2, n3)         = (0, 1, 0, 1)
```

tikz_boxed(*scale=1*, *boxsize=10*)

Return a tikzpicture of self included in a box.

INPUT:

- `scale` - number (default: 1), tikz scale
- `boxsize` - integer (default: 10), size of the box. If the width and height of the double square is less than the boxsize, then unit step are of size 1 and the (w_i) 8-tuple is added below the figure. Otherwise, if the width or height is larger than the boxsize, then the unit step are made smaller to fit the box and the (w_i) 8-tuple is not shown.

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(1))
sage: D.tikz_boxed()
\begin{tabular}{c}
\begin{tikzpicture}
[scale=1]
\filldraw[-to, very thick, draw=black, fill=black!20] (0.000,
0.000) -- (0.000, -1.00) -- (-1.00, -1.00) -- (-1.00, -2.00)
-- (0.000, -2.00) -- (0.000, -3.00) -- (1.00, -3.00) -- (1.00,
-2.00) -- (2.00, -2.00) -- (2.00, -1.00) -- (1.00, -1.00) --
(1.00, 0.000) -- (0.000, 0.000);
\node[first] at (0.0000, 0.0000) {};
\node[first] at (-1.000, -2.000) {};
\node[first] at (1.000, -3.000) {};
\node[first] at (2.000, -1.000) {};
\node[second] at (-1.000, -1.000) {};
\node[second] at (0.0000, -3.000) {};
\node[second] at (2.000, -2.000) {};
\node[second] at (1.000, 0.0000) {};
\end{tikzpicture}
\\
$({\bf 32},{\bf 3},{\bf 03},{\bf 0},$ \\
$\phantom{((}{\bf 10},{\bf 1},{\bf 21},{\bf 2})$ \\
\end{tabular}
```

Smaller boxsize:

```
sage: D.tikz_boxed(boxsize=1.5)
\begin{tikzpicture}
[scale=1]
\filldraw[-to, very thick, draw=black, fill=black!20] (0.000, 0.000) --
(0.000, -0.500) -- (-0.500, -0.500) -- (-0.500, -1.00) -- (0.000, -1.00) --
(0.000, -1.50) -- (0.500, -1.50) -- (0.500, -1.00) -- (1.00, -1.00) --
(1.00, -0.500) -- (0.500, -0.500) -- (0.500, 0.000) -- (0.000, 0.000);
\node[first] at (0.0000, 0.0000) {};
```

```
\node[first] at (-0.5000, -1.000) {};
\node[first] at (0.5000, -1.500) {};
\node[first] at (1.000, -0.5000) {};
\node[second] at (-0.5000, -0.5000) {};
\node[second] at (0.0000, -1.500) {};
\node[second] at (1.000, -1.000) {};
\node[second] at (0.5000, 0.0000) {};
\end{tikzpicture}
```

tikz_commutative_diagram(*tile*, *N=1*, *scale=(1, 1)*, *labels=True*, *newcommand=True*)

Return a tikz commutative diagram for the composition.

INPUT:

- `tile` - WordMorphism, a square tile
- `N` - integer (default:`1`), length of the diagram
- `scale` - tuple of number (default:`(1,1)`), one for each line
- `labels` - arrow labels (default:`True`). It may take the following values:
  - `True` - prints TRIM, SWAP, etc.
  - `'T'` - prints T_i, etc.
  - `False` - print nothing
- `newcommand` - bool (default: `True`), whether newcommand which defines `\SWAP`, `\TRIM`, etc.

EXAMPLES:

The following command creates the tikz code for Figure 16 in [BGL2012]:

```
sage: fibo2 = words.fibonacci_tile(2)
sage: S = WordMorphism({0:[0,0],1:[1,0,1],2:[2,2],3:[3,2,3]}, codomain=fibo2.parent())
sage: cfibo2 = DoubleSquare(fibo2)
sage: options = dict(tile=S,N=3,scale=(0.25,0.15),labels=True,newcommand=True)
sage: s = cfibo2.tikz_commutative_diagram(**options)     # long time (2s)
sage: s                                                  # long time
\newcommand{\TRIM}{\textsc{trim}}
\newcommand{\EXTEND}{\textsc{extend}}
\newcommand{\SWAP}{\textsc{swap}}
\newcommand{\SHIFT}{\textsc{shift}}
\newcommand{\REVERSE}{\textsc{reverse}}
\begin{tikzpicture}
[first/.style={circle,draw=black,fill=black, inner sep=0pt, minimum size=3pt},
second/.style={circle,draw=black,fill=white, inner sep=0pt, minimum size=3pt}]
\node (q0) at (0, 0) {\begin{tikzpicture}
[scale=0.250000000000000]
...
\end{tikzpicture}};
\path[thick, ->] (r0) edge node[midway, rectangle, fill=white, rotate=90] {$\SWAP_1$} (r1);
\path[thick, ->] (r1) edge node[midway, rectangle, fill=white, rotate=90] {$\TRIM_1$} (r2);
\path[thick, ->] (r2) edge node[midway, rectangle, fill=white, rotate=90] {$\TRIM_3$} (r3);
\path[thick, ->] (q0) edge node[midway, left] {$\varphi$} (r0);
\path[thick, ->] (q1) edge node[midway, left] {$\varphi$} (r1);
\path[thick, ->] (q2) edge node[midway, left] {$\varphi$} (r2);
\path[thick, ->] (q3) edge node[midway, left] {$\varphi$} (r3);
\end{tikzpicture}
```

tikz_reduction(*scale=1*, *ncols=3*, *gridstep=5*, *labels=True*, *newcommand=True*)

INPUT:

- `scale` - number
- `ncols` - integer, number of columns displaying the reduction

- `gridstep` - number (default: `5`), the gridstep for the snake node positions
- `labels` - arrow labels (default:`True`). It may take the following values:
  - `True` - prints TRIM, SWAP, etc.
  - `'T'` - prints T_i, etc.
  - `False` - print nothing
- `newcommand` - bool (default: `True`), whether newcommand which defines `\SWAP`, `\TRIM`, etc.

EXAMPLES:

```
sage: fibo2 = words.fibonacci_tile(2)
sage: cfibo2 = DoubleSquare(fibo2)
sage: s = cfibo2.tikz_reduction(scale=0.5,ncols=4,labels=True)
sage: s
\newcommand{\TRIM}{\textsc{trim}}
\newcommand{\EXTEND}{\textsc{extend}}
\newcommand{\SWAP}{\textsc{swap}}
\newcommand{\SHIFT}{\textsc{shift}}
\newcommand{\REVERSE}{\textsc{reverse}}
\begin{tikzpicture}
[first/.style={circle,draw=black,fill=black, inner sep=0pt, minimum size=3pt},
second/.style={circle,draw=black,fill=white, inner sep=0pt, minimum size=3pt}]
\node (q0) at (0, 0) {
\begin{tikzpicture}
[scale=0.500000000000000]
...
\end{tikzpicture}
\\
$(\varepsilon,{\bf 3},\varepsilon,{\bf 2},$ \\
$\phantom{(}\varepsilon,{\bf 1},\varepsilon,{\bf 0})$ \\
\end{tabular}
};
\path[->] (q0) edge node[midway, rectangle, fill=white, rotate=90] {$\SWAP_1$} (q1);
\path[->] (q1) edge node[midway, rectangle, fill=white, rotate=90] {$\TRIM_1$} (q2);
\path[->] (q2) edge node[midway, rectangle, fill=white, rotate=90] {$\TRIM_3$} (q3);
\path[->] (q3) edge node[midway, rectangle, fill=white] {$\TRIM_0$} (q4);
\path[->] (q4) edge node[midway, rectangle, fill=white, rotate=90] {$\TRIM_2$} (q5);
\end{tikzpicture}

sage: S = WordMorphism({0:[0,0],1:[1,0,1],2:[2,2],3:[3,2,3]}, codomain=fibo2.parent())
sage: cSfibo2 = cfibo2.apply_morphism(S)
sage: s = cSfibo2.tikz_reduction(scale=0.15,ncols=4,labels='T')
sage: s
\newcommand{\TRIM}{\textsc{trim}}
\newcommand{\EXTEND}{\textsc{extend}}
\newcommand{\SWAP}{\textsc{swap}}
\newcommand{\SHIFT}{\textsc{shift}}
\newcommand{\REVERSE}{\textsc{reverse}}
\begin{tikzpicture}
[first/.style={circle,draw=black,fill=black, inner sep=0pt, minimum size=3pt},
second/.style={circle,draw=black,fill=white, inner sep=0pt, minimum size=3pt}]
\node (q0) at (0, 0) {
\begin{tikzpicture}
[scale=0.150000000000000]
...
\end{tikzpicture}
\\
$(\varepsilon,{\bf 323},\varepsilon,{\bf 22},$ \\
$\phantom{(}\varepsilon,{\bf 101},\varepsilon,{\bf 00})$ \\
\end{tabular}
};
\path[thick, ->] (q0) edge node[midway, above] {$T_1$} (q1);
\path[thick, ->] (q1) edge node[midway, above] {$T_2$} (q2);
\path[thick, ->] (q2) edge node[midway, above] {$T_3$} (q3);
\path[thick, ->] (q3) edge node[midway, above] {$T_4$} (q4);
```

```
\path[thick, ->] (q4) edge node[midway, above] {$T_5$} (q5);
\end{tikzpicture}
```

tikz_trajectory(*step=1*, *arrow='->'*)

Returns a tikz string describing the double square induced by this configuration together with its factorization points

The factorization points respectively get the tikz attribute 'first' and 'second' so that when including it in a tikzpicture environment, it is possible to modify the way those points appear.

INPUT:

- `step` - integer (default: 1)
- `arrow` - string (default: ->), tikz arrow shape

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(1))
sage: D.tikz_trajectory()
\filldraw[->, very thick, draw=black, fill=black!20] (0.000, 0.000)
-- (0.000, -1.00) -- (-1.00, -1.00) -- (-1.00, -2.00) -- (0.000, -2.00) --
(0.000, -3.00) -- (1.00, -3.00) -- (1.00, -2.00) -- (2.00, -2.00) -- (2.00,
-1.00) -- (1.00, -1.00) -- (1.00, 0.000) -- (0.000, 0.000); \node[first] at
(0.0000, 0.0000) {};
\node[first] at (-1.000, -2.000) {};
\node[first] at (1.000, -3.000) {};
\node[first] at (2.000, -1.000) {};
\node[second] at (-1.000, -1.000) {};
\node[second] at (0.0000, -3.000) {};
\node[second] at (2.000, -2.000) {};
\node[second] at (1.000, 0.0000) {};
```

trim(*i*)

Apply $TRIM_i$ on self.

This removes a period of length d_i to w_i and w_{i+4}.

INPUT:

- `i` - integer

EXAMPLES:

```
sage: D = DoubleSquare((3,6,3,2))
sage: D.trim(1)
Double Square Tile
  w0 = 212   w4 = 030
  w1 =       w5 =
  w2 = 303   w6 = 121
  w3 = 03    w7 = 21
(|w0|, |w1|, |w2|, |w3|) = (3, 0, 3, 2)
(d0, d1, d2, d3)         = (2, 6, 2, 6)
(n0, n1, n2, n3)         = (1, 0, 1, 0)

sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D.extend(3).trim(3)
Double Square Tile
  w0 = 32303010   w4 = 10121232
  w1 = 30323      w5 = 12101
  w2 = 21232303   w6 = 03010121
  w3 = 23212      w7 = 01030
(|w0|, |w1|, |w2|, |w3|) = (8, 5, 8, 5)
(d0, d1, d2, d3)         = (10, 16, 10, 16)
(n0, n1, n2, n3)         = (0, 0, 0, 0)
```

turning_number()

Return the turning number of self.

INPUT:

- `self` - double square defined on the alphabet of integers $\{0, 1, 2, 3\}$

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(1))
sage: D.turning_number()
1
sage: D.reverse().turning_number()
-1
```

Turning number of a degenerate double square:

```
sage: D = DoubleSquare(([],[0],[1,0],[1]))
sage: D.turning_number()
1
```

Turning number of a singular double square:

```
sage: D = DoubleSquare(([],[0,3,0,1,0],[],[1,0,1,1]))
sage: D.turning_number()
1
```

Turning number of a flat double square:

```
sage: D = DoubleSquare(([],[],[0,1,0,1],[0,1]))
sage: D.turning_number()
0
```

u(i)

Return the word u_i.

The word u_i is the unique word such that $w_i = (u_i * v_i)^{n_i} u_i$ where $0 \leq |u_i| < d_i$.

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D.u(1)
Path: 30323
```

```
sage: rot180 = WordMorphism('0->2,2->0,3->1,1->3')
sage: steps = {'0':(1,0), '1':(0,1), '2':(-1,0), '3': (0,-1)}
sage: D = DoubleSquare(('','03010','','1011'), rot180, steps)
sage: D.u(0)
word:
sage: D.u(1)
Traceback (most recent call last):
...
ValueError: u_1 is not defined when d_1 == 0
```

v(i)

Return the word v_i.

The word v_i is the unique word such that $w_i = (u_i * v_i)^{n_i} u_i$ where $0 \leq |u_i| < d_i$, $\hat{w_{i-3}} w_{i-1} = u_i v_i$ and $0 < |u_i| \leq d_i$.

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D.v(1)
Path: 21232303010
```

```
sage: rot180 = WordMorphism('0->2,2->0,3->1,1->3')
sage: steps = {'0':(1,0), '1':(0,1), '2':(-1,0), '3': (0,-1)}
sage: D = DoubleSquare(('','03010','','1011'), rot180, steps)
sage: D.v(0)
word: 030103323
sage: D.v(1)
Traceback (most recent call last):
...
ValueError: v_1 is not defined when d_1 == 0
```

verify_definition()

Checks that the input verifies the definition.

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D.verify_definition()
```

```
sage: DoubleSquare(([],[0],[0,1,0,1],[0,1]))
Traceback (most recent call last):
...
AssertionError: wiwi+1 = hat(wi+4,wi+5) is not verified for i=1
```

w(i)

Return the factor w_i

This corresponds to the new definition of configuration (solution).

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(1))
sage: [D.w(i) for i in range(8)]
[Path: 32, Path: 3, Path: 03, Path: 0, Path: 10, Path: 1, Path: 21, Path: 2]
```

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: [D.w(i) for i in range(8)]
[Path: 32303010, Path: 30323, Path: 21232303, Path: 23212, Path: 10121232, Path: 12101, Path: 03010121, Path: 01030]
```

width()

Returns the width of this polyomino, i.e. the difference between its rightmost and leftmost coordinates

EXAMPLES:

```
sage: D = DoubleSquare(words.fibonacci_tile(2))
sage: D.width()
9
```

```
sage: D = DoubleSquare((34,21,34,21))
sage: D.width()
23
```

slabbe.double_square_tile.**christoffel_tile**(p, q)

Returns the (p, q) Christoffel Tile [BBGL2011].

EXAMPLES:

sage: christoffel_tile(7,9) Path: 0301030101030101030101030103010301030101030103... sage: christoffel_tile(9,7) Path: 0301010301010301010301010103010301030101030... sage: christoffel_tile(2,3) Path: 030103010103010121232123232123232123232123232123 sage: christoffel_tile(0,1) Path: 03012123 sage: print christoffel_tile(4,5) 030103010103010103010103010121232123232123232123232123232123

slabbe.double_square_tile.**double_square_from_boundary_word**(ds)

Creates a double square object from the boundary word of a double square tile.

INPUT:

- ds - word, the boundary of a double square. The parent alphabet is assumed to be in the order : East, North, West, South.

OUTPUT:

- tuple - tuple of 8 words over the alphabet A
- WordMorphism, involution on the alphabet A and representing a rotation of 180 degrees.
- dict - mapping letters of A to steps in the plane.

EXAMPLES:

```
sage: from slabbe.double_square_tile import double_square_from_boundary_word
sage: fibo = words.fibonacci_tile
sage: W, rot180, steps = double_square_from_boundary_word(fibo(1))
sage: map(len, W)
[2, 1, 2, 1, 2, 1, 2, 1]
sage: W, rot180, steps = double_square_from_boundary_word(fibo(2))
sage: map(len, W)
[8, 5, 8, 5, 8, 5, 8, 5]
sage: W, rot180, steps = double_square_from_boundary_word(fibo(3))  # long time (6s)
sage: map(len, W)                                                  # long time
[34, 21, 34, 21, 34, 21, 34, 21]
sage: rot180                                                       # long time
WordMorphism: 0->2, 1->3, 2->0, 3->1
```

slabbe.double_square_tile.**double_square_from_four_integers**(*l0*, *l1*, *l2*, *l3*)

Creates a double square from the lengths of the w_i.

INPUT:

- l0 - integer, length of w_0
- l1 - integer, length of w_1
- l2 - integer, length of w_2
- l3 - integer, length of w_3

OUTPUT:

- tuple - tuple of 8 words over alphabet A
- WordMorphism, involution on the alphabet A and representing a rotation of 180 degrees.
- dict - mapping letters of A to steps in the plane.

EXAMPLES:

```
sage: from slabbe.double_square_tile import double_square_from_four_integers
sage: w,rot180,steps = double_square_from_four_integers(2,1,2,1)
sage: w
(Path: 21, Path: 2, Path: 32, Path: 3, Path: 03, Path: 0, Path: 10, Path: 1)
sage: rot180
WordMorphism: 0->2, 1->3, 2->0, 3->1
sage: sorted(steps.items())
[(0, (1, 0)), (1, (0, 1)), (2, (-1, 0)), (3, (0, -1))]
```

If the input integers do not define a double square uniquely, the alphabet might be larger than 8:

```
sage: w,rot180,steps = double_square_from_four_integers(4,2,4,2)
sage: w
(Path: 7601,
 Path: 76,
 Path: 5476,
 Path: 54,
```

```
 Path: 2354,
 Path: 23,
 Path: 0123,
 Path: 01)
sage: rot180
WordMorphism: 0->4, 1->5, 2->6, 3->7, 4->0, 5->1, 6->2, 7->3
sage: sorted(steps.items())
[(0, (1, 0)),
 (1, (1/2*sqrt(2), 1/2*sqrt(2))),
 (2, (0, 1)),
 (3, (-1/2*sqrt(2), 1/2*sqrt(2))),
 (4, (-1, 0)),
 (5, (-1/2*sqrt(2), -1/2*sqrt(2))),
 (6, (0, -1)),
 (7, (1/2*sqrt(2), -1/2*sqrt(2)))]
```

slabbe.double_square_tile.**figure_11_BGL2012**(*scale=0.5*, *boxsize=10*, *newcommand=True*)

Return the tikz code of the Figure 11 for the article [BGL2012].

INPUT:

- `scale` - number (default: `0.5`), tikz scale
- `boxsize` - integer (default: `10`), size of box the double squares must fit in
- `newcommand` - bool (default: `True`), whether to include latex newcommand for TRIM, EXTEND and SWAP

EXAMPLES:

```
sage: from slabbe.double_square_tile import figure_11_BGL2012
sage: s = figure_11_BGL2012()
sage: s
\newcommand{\TRIM}{\textsc{trim}}
\newcommand{\EXTEND}{\textsc{extend}}
\newcommand{\SWAP}{\textsc{swap}}
\begin{tikzpicture}
[first/.style={circle,draw=black,fill=black, inner sep=0pt, minimum size=3pt},
second/.style={circle,draw=black,fill=white, inner sep=0pt, minimum size=3pt},
>=latex,
node distance=3cm]
\node (A) at (15,0)
{
\begin{tabular}{c}
\begin{tikzpicture}
[scale=0.5]
...
\end{tikzpicture}
};
\path[<-] (A) edge node[midway, rectangle, fill=white] {$\TRIM_2$} (B);
\path[<-] (B) edge node[midway, rectangle, fill=white] {$\TRIM_0$} (C);
\path[<-] (C) edge node[midway, rectangle, fill=white] {$\TRIM_1$} (D);
\path[<-] (D) edge node[midway, rectangle, fill=white] {$\TRIM_3$} (E);
\path[<-] (E) edge node[midway, rectangle, fill=white] {$\SWAP_1$} (F);
\path[<-] (F) edge node[midway, rectangle, fill=white] {$\TRIM_1$} (G);
\path[<-] (G) edge node[midway, rectangle, fill=white,rotate=90] {$\TRIM_3$} (H);
\path[<-] (H) edge node[midway, rectangle, fill=white,rotate=90] {$\SWAP_1$} (I);
\path[<-] (D) edge node[midway, rectangle, fill=white] {$\TRIM_2$} (E2);
\end{tikzpicture}
```

slabbe.double_square_tile.**find_square_factorisation**(*ds*, *factorisation=None*, *alternate=True*)

Return a square factorisation of the double square ds, distinct from the given factorisation.

INPUT:

- `ds` - word, the boundary word of a square tile

- `factorisation` - tuple (default: `None`), a known factorisation
- `alternate` - bool (default: `True`), if True the search for the second factorisation is restricted to those who alternates with the first factorisation

OUTPUT:

tuple of four positions of a square factorisation

EXAMPLES:

```
sage: from slabbe.double_square_tile import find_square_factorisation
sage: find_square_factorisation(words.fibonacci_tile(0))
(0, 1, 2, 3)
sage: find_square_factorisation(words.fibonacci_tile(1))
(0, 3, 6, 9)
sage: find_square_factorisation(words.fibonacci_tile(2))
(0, 13, 26, 39)
sage: find_square_factorisation(words.fibonacci_tile(3))
(0, 55, 110, 165)

sage: f = find_square_factorisation(words.fibonacci_tile(3))
sage: f
(0, 55, 110, 165)
sage: find_square_factorisation(words.fibonacci_tile(3),f)         # long time (6s)
(34, 89, 144, 199)
sage: find_square_factorisation(words.fibonacci_tile(3),f,False)   # long time (11s)
(34, 89, 144, 199)

sage: find_square_factorisation(christoffel_tile(4,5))
(0, 7, 28, 35)
sage: find_square_factorisation(christoffel_tile(4,5),_)
(2, 27, 30, 55)
```

TESTS:

```
sage: find_square_factorisation(Words('abcd')('aaaaaa'))
Traceback (most recent call last):
...
ValueError: no square factorization found
sage: find_square_factorisation(Words('abcd')('aaaaaa'),(1,2,3,4))
Traceback (most recent call last):
...
ValueError: no second square factorization found
```

`slabbe.double_square_tile.`**`snake`**(*i*, *ncols=2*)

Return the coordinate of the ith node of a snake.

This is used for the tikz drawing of a double square reduction.

INPUT:

- `i` - integer, the ith node
- `ncols` - integer (default 2), number of columns

EXAMPLES:

```
sage: from slabbe.double_square_tile import snake
sage: for i in range(8): snake(i, 3)
(0, 0)
(1, 0)
(2, 0)
(2, -1)
(1, -1)
(0, -1)
(0, -2)
(1, -2)
```

slabbe.double_square_tile.**triple_square_example**(*i*)

Return a triple square factorisation example.

These words having three square factorisations were provided by Xavier Provençal.

INPUT:

- `i` - integer, accepted values are 1, 2 or 3.

EXAMPLES:

```
sage: from slabbe.double_square_tile import triple_square_example
sage: triple_square_example(1)
Path: abaBAbababABababaBAbabABABabABABabABAB
sage: triple_square_example(2)
Path: abaBABaabaBABaabaBABaabABAAbabABAAbabABA...
sage: triple_square_example(3)
Path: aabAAbaabAAbaabAAbaaBAABaaBAABaaBAAB
```

Triple square tile do not exist. Hence the example provided by Xavier Provençal can not be the boundary word of a tile. One can see it by ploting it or by the fact that the turning number is zero:

```
sage: D = DoubleSquare(triple_square_example(1))
sage: D
Double Square Tile
  w0 = a          w4 = a
  w1 = baBA       w5 = bABA
  w2 = babaB      w6 = BabAB
  w3 = AbabaBAb   w7 = ABabABAB
(|w0|, |w1|, |w2|, |w3|) = (1, 4, 5, 8)
(d0, d1, d2, d3)         = (12, 6, 12, 6)
(n0, n1, n2, n3)         = (0, 0, 0, 1)
sage: D.turning_number()
0
```

CHAPTER

TWO

COMBINATORICS ON WORDS

2.1 Kolakoski Word

The classical Kolakoski sequence [K65] was first studied by Oldenburger [O39], where it appears as the unique solution to the problem of a trajectory on the alphabet $\{1, 2\}$ which is identical to its exponent trajectory.

See http://en.wikipedia.org/wiki/Kolakoski_sequence

It uses cython implementation inspired from the 10 lines of C code written by Dominique Bernardi and shared during Sage Days 28 in Orsay, France, in January 2011. The alphabet must be `(1,2)`.

EXAMPLES:

```
sage: K = KolakoskiWord()
sage: K
word: 122112122122112112212112122112112122122 1...
```

The cython implementation is much faster than the python one:

```
sage: K = KolakoskiWord()
sage: K[10^6]      # takes 0.02 seconds
2
sage: K = words.KolakoskiWord()
sage: K[10^6]      # not tested : takes too long
2
```

TESTS:

We make sure both implementation correspond for the prefix of length 100:

```
sage: Kb = KolakoskiWord()
sage: Kp = words.KolakoskiWord()
sage: Kp[:100] == Kb[:100]
True
```

REFERENCES:

AUTHORS:

- Sébastien Labbé (February 2011) - Added a Cython implementation which was easily translated from the 10 lines of C code written by Dominique Bernardi and shared during Sage Days 28 in Orsay. Needs review at #13346 since too long time...

class slabbe.kolakoski_word.**KolakoskiWord**(*parent=None*)

Bases: slabbe.kolakoski_word_pyx.WordDatatype_Kolakoski, sage.combinat.words.infinite_word.InfiniteWo

Constructor. See documentation of WordDatatype_Kolakoski for more details.

EXAMPLES:

```
sage: K = KolakoskiWord()
sage: K
word: 122112122122112112212112122112112122122 1...
```

TESTS:

Pickle is supported:

```
sage: K = KolakoskiWord()
sage: loads(dumps(K))
word: 122112122122112112212112122112112122122 1...
```

class slabbe.kolakoski_word_pyx.**WordDatatype_Kolakoski**

Bases: object

Word datatype class for the Kolakoski word.

This is a cython implementation inspired from the 10 lines of C code written by Dominique Bernardi and shared during Sage Days 28 in Orsay, France, in January 2011.

INPUT:

- parent - the parent

EXAMPLES:

```
sage: from slabbe.kolakoski_word_pyx import WordDatatype_Kolakoski
sage: parent = Words([1,2])
sage: K = WordDatatype_Kolakoski(parent)
sage: K
<slabbe.kolakoski_word_pyx.WordDatatype_Kolakoski object at ...>
sage: K[0]
1
sage: K[1]
2
```

2.2 Factor complexity and Bispecial Extension Type

This module was developped for the article on the factor complexity of infinite sequences genereated by substitutions written with Valérie Berthé [BL2014].

EXAMPLES:

The extension type of an ordinary bispecial factor:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, alphabet=(1,2,3))
sage: E
  E(w)   1   2   3
   1             X
   2             X
   3     X   X   X
 m(w)=0, ordinary
sage: E.is_ordinaire()
True
```

Creation of a strong-weak pair of bispecial words from a neutral not ordinairy word:

```
sage: m = WordMorphism({1:[1,2,3],2:[2,3],3:[3]})
sage: E = ExtensionType1to1([(1,2),(2,3),(3,1),(3,2),(3,3)], [1,2,3])
sage: E
  E(w)   1   2   3
```

```
   1         X
   2             X
   3     X   X   X
 m(w)=0, not ord.
sage: E1, E2 = E.apply(m)
sage: E1
  E(3w)   1   2   3
    1
    2         X   X
    3     X   X   X
 m(w)=1, not ord.
sage: E2
  E(23w)   1   2   3
    1          X
    2
    3              X
 m(w)=-1, not ord.
```

TODO:

- use __classcall_private__ stuff for ExtensionType ?
- rename ExtensionType2to1 to ExtendedExtensionType ?
- export tikz to pdf using view instead of tikz2pdf ?
- fix bug of apply for ExtensionType2to1 when the word appears in the image of a letter
- add the bispecial word to the attribute of extension type
- use this to compute the factor complexity function

class slabbe.bispecial_extension_type.**ExtensionType**

Bases: object

x.__init__(...) initializes x; see help(type(x)) for signature

cardinality()

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, [1,2,3])
sage: E.cardinality()
5
```

equivalence_class()

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:    2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: len(E.equivalence_class())
6
```

static from_factor(*bispecial*, *word*)

EXAMPLES:

```
sage: W = Words([0,1,2])
sage: ExtensionType.from_factor(W(), W([0,1,1,2,0]))
  E(w)   0   1   2
   0         X
   1         X   X
   2     X
 m(w)=-1, not ord.
```

static from_factor2(*bispecial*, *word*)

EXAMPLES:

```
sage: W = Words([0,1,2])
sage: ExtensionType.from_factor2(W(), W([0,1,1,2,0]))
  E(w)   0   1   2
   0         X
   1         X   X
   2     X
 m(w)=-1, not ord.
```

static from_morphism(*m*)

Return the extension type of the empty word in the language defined by the image of the free monoid under the morphism m.

INPUT:

- m - endomorphim

EXAMPLES:

```
sage: ar = WordMorphism({1:[1,3],2:[2,3],3:[3]})
sage: ExtensionType.from_morphism(ar)
  E(w)   1   2   3
   1             X
   2             X
   3     X   X   X
 m(w)=0, ordinary
```

```
sage: p = WordMorphism({1:[1,2,3],2:[2,3],3:[3]})
sage: ExtensionType.from_morphism(p)
  E(w)   1   2   3
   1         X
   2             X
   3     X   X   X
 m(w)=0, not ord.
```

```
sage: b12 = WordMorphism({1:[1,2],2:[2],3:[3]})
sage: ExtensionType.from_morphism(b12)
  E(w)   1   2   3
   1         X
   2     X   X   X
   3     X   X   X
 m(w)=2, not ord.
```

image(*m*)

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:          2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: b23 = WordMorphism({1:[1],2:[2,3],3:[3]})
sage: E.image(b23)
  E(w)   1   2   3
   31        X
   12            X
   32            X
   23    X   X   X
   33    X
 m(w)=0, not ord., empty
```

is_bispecial()

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:    2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
```

```
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E.is_bispecial()
True
```

is_empty()

Return whether the word associated to this extension type is empty.

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, alphabet=(1,2,3))
sage: E.is_empty()
False

sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:          2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E.is_empty()
True

sage: E = ExtensionType2to1(L, (1,2,3), empty=False)
sage: E.is_empty()
False
```

is_equivalent(*other*)

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:    2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E.is_equivalent(E)
True
```

is_neutral()

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, [1,2,3])
sage: E.is_neutral()
True
```

is_ordinaire()

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, alphabet=(1,2,3))
sage: E.is_ordinaire()
True
```

left_extensions()

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, alphabet=(1,2,3))
sage: E.left_extensions()
set([1, 2, 3])
```

left_right_projection()

EXAMPLES:

```
sage: L = [(1,2), (2,2), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, alphabet=(1,2,3))
sage: E.left_right_projection()
(Counter({3: 3, 1: 1, 2: 1}), Counter({2: 3, 1: 1, 3: 1}))
```

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:          2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: left, right = E.left_right_projection()
sage: sorted(left.iteritems())
[(word: 12, 3), (word: 21, 1), (word: 22, 1), (word: 23, 1), (word: 31, 1)]
sage: sorted(right.iteritems())
[(word: 1, 3), (word: 2, 3), (word: 3, 1)]
```

left_valence()

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, [1,2,3])
sage: E.left_valence()
3
```

life_graph(*substitutions*, *substitutions_dict=None*)

Return the graph of extension types generated under a sequence of substitutions.

INPUT:

- `substitutions` - list of substitutions keys
- `substitutions_dict` - dict of substitutions, if None then it gets replaced by `common_substitutions_dict` defined in the module.

EXAMPLES:

From an ordinaire word:

```
sage: e = ExtensionType1to1([(1,3),(2,3),(3,1),(3,2),(3,3)], [1,2,3])
sage: e.life_graph(['p23'])
Looped multi-digraph on 2 vertices
sage: e.life_graph(['p32'])
Looped multi-digraph on 3 vertices
sage: e.life_graph(['p32','p13','ar2'])
Looped multi-digraph on 5 vertices
```

2to1:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:     2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E.life_graph(['b12','b21','b12'])
Looped multi-digraph on 8 vertices
```

life_graph_save_tikz(*filename*, *substitutions*, ***kwds*)

INPUT:

- "filename" - string
- "format" - string, default: 'tkz_graph' – either 'dot2tex' or 'tkz_graph'.

If format is 'dot2tex', then all the LaTeX generation will be delegated to "dot2tex" (which must be installed).

For the 'dot2tex' format, the possible option names and associated values are given below:

- "prog" – the program used for the layout. It must be a string corresponding to one of the software of the graphviz suite: 'dot', 'neato', 'twopi', 'circo' or 'fdp'.

See this for more options:

```
sage: g = graphs.PetersenGraph()
sage: opts = g.latex_options()
sage: opts.set_option?        # not tested
```

EXAMPLES:

```
sage: e = ExtensionType1to1([(1,3),(2,3),(3,1),(3,2),(3,3)], [1,2,3])
sage: e.life_graph_save_tikz('a.tikz', ['p32','p13','ar2'])    # not tested
```

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:    2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E.life_graph_save_tikz('b.tikz', ['b12','b21','b12']) # not tested
Creation of file b.tikz
Using template '/Users/slabbe/.tikz2pdf.tex'.
tikz2pdf: calling pdflatex...
tikz2pdf: Output written to 'b.pdf'.
```

multiplicity()

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, [1,2,3])
sage: E.multiplicity()
0
```

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:    2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E.multiplicity()
0
```

right_extensions()

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, alphabet=(1,2,3))
sage: E.right_extensions()
set([1, 2, 3])
```

right_valence()

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, [1,2,3])
sage: E.right_valence()
3
```

class slabbe.bispecial_extension_type.**ExtensionType1to1**(*L*, *alphabet*, *chignons=('', '')*)

Bases: slabbe.bispecial_extension_type.ExtensionType

INPUT:

- `L` - list of pairs of letters
- `alphabet` - the alphabet
- `chignons` - optional (default: None), pair of words added to the left and to the right of the image of the previous bispecial

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, alphabet=(1,2,3))
sage: E
```

```
  E(w)   1   2   3
   1             X
   2             X
   3     X   X   X
 m(w)=0, ordinary
```

With chignons:

```
sage: E = ExtensionType1to1(L, [1,2,3], ('a','b'))
sage: E
  E(awb)   1   2   3
    1              X
    2              X
    3      X   X   X
 m(w)=0, ordinary
```

apply(*m*)

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, [1,2,3])
sage: E
  E(w)   1   2   3
   1             X
   2             X
   3     X   X   X
 m(w)=0, ordinary
sage: ar = WordMorphism({1:[1,3],2:[2,3],3:[3]})
sage: E.apply(ar)
(  E(3w)   1   2   3
    1             X
    2             X
    3     X   X   X
 m(w)=0, ordinary,)

sage: ar = WordMorphism({1:[3,1],2:[3,2],3:[3]})
sage: E.apply(ar)
(  E(w3)   1   2   3
    1             X
    2             X
    3     X   X   X
 m(w)=0, ordinary,)
```

Creation of a pair of ordinaire bispecial words from an **ordinaire** word:

```
sage: e = ExtensionType1to1([(1,3),(2,3),(3,1),(3,2),(3,3)], [1,2,3])
sage: p0 = WordMorphism({1:[1,2,3],2:[2,3],3:[3]})
sage: e.apply(p0)
(  E(3w)   1   2   3
    1
    2             X
    3     X   X   X
 m(w)=0, ordinary,)
sage: p3 = WordMorphism({1:[1,3,2],2:[2],3:[3,2]})
sage: e.apply(p3)
(  E(2w)   1   2   3
    1
    2             X
    3     X   X   X
 m(w)=0, ordinary,
   E(32w)   1   2   3
    1              X
    2      X   X   X
    3
 m(w)=0, ordinary)
```

Creation of a strong-weak pair of bispecial words from a neutral **not ordinaire** word:

```
sage: p0 = WordMorphism({1:[1,2,3],2:[2,3],3:[3]})
sage: e = ExtensionType1to1([(1,2),(2,3),(3,1),(3,2),(3,3)], [1,2,3])
sage: e.apply(p0)
(  E(3w)   1   2   3
    1
    2         X   X
    3     X   X   X
 m(w)=1, not ord.,
   E(23w)   1   2   3
    1          X
    2
    3              X
 m(w)=-1, not ord.)
```

Creation of a pair of ordinaire bispecial words from an **not ordinaire** word:

```
sage: p1 = WordMorphism({1:[1,2],2:[2],3:[3,1,2]})
sage: e = ExtensionType1to1([(1,2),(2,3),(3,1),(3,2),(3,3)], [1,2,3])
sage: e.apply(p1)
(  E(2w)   1   2   3
    1     X   X   X
    2             X
    3
 m(w)=0, ordinary,
   E(12w)   1   2   3
    1
    2          X
    3      X   X   X
 m(w)=0, ordinary)
```

This result is now fixed:

```
sage: e = ExtensionType1to1([(1,2), (3,3)], [1,2,3])
sage: p3 = WordMorphism({1:[1,3,2],2:[2],3:[3,2]})
sage: e.apply(p3)
(  E(32w)   1   2   3
    1          X
    2              X
    3
 m(w)=-1, not ord.,)

sage: e = ExtensionType1to1([(2,2),(2,3),(3,1),(3,2),(3,3)], [1,2,3])
sage: e.apply(p3)
(  E(2w)   1   2   3
    1
    2         X   X
    3     X   X   X
 m(w)=1, not ord.,)
```

This result is now fixed:

```
sage: e = ExtensionType1to1([(2,2),(2,3),(3,1),(3,2),(3,3)], [1,2,3])
sage: p2 = WordMorphism({1:[1],2:[2,3,1],3:[3,1]})
sage: e.apply(p2)
(  E(31w)   1   2   3
    1      X   X   X
    2          X   X
    3
 m(w)=1, not ord.,)

sage: e = ExtensionType1to1([(1,2),(3,3)], [1,2,3])
sage: e.apply(p2)
(  E(1w)   1   2   3
    1         X
    2
```

```
    3             X
 m(w)=-1, not ord.,)
```

TESTS:

```
sage: L = [(1,3)]
sage: E = ExtensionType1to1(L, [1,2,3])
sage: E
  E(w)   1   2   3
   1             X
   2
   3
 m(w)=0, not ord.
sage: ar = WordMorphism({1:[1,3],2:[2,3],3:[3]})
sage: E.apply(ar)
()
```

POSSIBLE BUG:

```
sage: b23 = WordMorphism({1:[1],2:[2,3],3:[3]})
sage: b13 = WordMorphism({1:[1,3],2:[2],3:[3]})
sage: b31 = WordMorphism({1:[1],2:[2],3:[3,1]})
sage: e = ExtensionType.from_morphism(b23)
sage: r = e.apply(b23)[0]
sage: r.apply(b13)
()
sage: r.apply(b31)
()
```

cardinality()

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, [1,2,3])
sage: E.cardinality()
5
```

chignons_multiplicity_tuple()

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, [1,2,3], ('a', 'b'))
sage: E.chignons_multiplicity_tuple()
('a', 'b', 0)
```

is_ordinaire()

EXAMPLES:

ordinary:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, alphabet=(1,2,3))
sage: E
  E(w)   1   2   3
   1             X
   2             X
   3     X   X   X
 m(w)=0, ordinary
sage: E.is_ordinaire()
True
```

strong:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3), (1,1)]
sage: E = ExtensionType1to1(L, alphabet=(1,2,3))
```

```
sage: E.is_ordinaire()
False
```

neutral but not ordinary:

```
sage: L = [(1,1), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, alphabet=(1,2,3))
sage: E
  E(w)   1   2   3
   1     X
   2             X
   3     X   X   X
 m(w)=0, not ord.
sage: E.is_neutral()
True
sage: E.is_ordinaire()
False
```

not neutral, not ordinaire:

```
sage: L = [(1,1), (2,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, alphabet=(1,2,3))
sage: E
  E(w)   1   2   3
   1     X
   2     X
   3         X   X
 m(w)=-1, not ord.
sage: E.is_neutral()
False
sage: E.is_ordinaire()
False
```

left_extensions()

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, [1,2,3])
sage: E.left_extensions()
set([1, 2, 3])
```

right_extensions()

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, [1,2,3])
sage: E.right_extensions()
set([1, 2, 3])
```

table()

return a table representation of self.

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, alphabet=(1,2,3))
sage: E.table()
  E(w)   1   2   3
   1             X
   2             X
   3     X   X   X
```

```
sage: E = ExtensionType1to1(L, alphabet=(1,2,3), chignons=('a', 'b'))
sage: E.table()
  E(awb)   1   2   3
    1              X
```

```
2             X
3     X   X   X
```

class slabbe.bispecial_extension_type.**ExtensionType2to1**(*L*, *alphabet*, *chignons=('', '')*, *factors_length_2=None*, *empty=None*)

Bases: slabbe.bispecial_extension_type.ExtensionType

Generalized to words.

INPUT:

- `L` - list of pairs of *words*
- `alphabet` - the alphabet
- `chignons` - optional (default: None), pair of words added to the left and to the right of the image of the previous bispecial
- `factors_length_2` - list of factors of length 2. If None, they are computed from the provided extension assuming the bispecial factor is *empty*.
- `empty` - bool, (optional, default: None), if None, then it is computed from the chignons and takes value True iff the chignons are empyt.

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:          2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E
  E(w)   1   2   3
   21        X
   31        X
   12    X   X   X
   22    X
   23    X
m(w)=0, not ord., empty
```

apply(*m*, *l=2*, *r=1*)

The code works for Brun here because we take length 2 on the left and length 1 on the right.

On utilise les facteurs de longueur 2 pour completer l'info qui peut manquer.

TODO: bien corriger les facteurs de longueurs 2 de l'image!!!

INPUT:

- `m` - substitution
- `l` - integer, length of left extension
- `r` - integer, length of right extension

OUTPUT:

list of Extension type of the bispecial images

POSSIBLE BUG:

```
sage: b23 = WordMorphism({1:[1],2:[2,3],3:[3]})
sage: b13 = WordMorphism({1:[1,3],2:[2],3:[3]})
sage: b31 = WordMorphism({1:[1],2:[2],3:[3,1]})
sage: e = ExtensionType.from_morphism(b23)
sage: r = e.apply(b23)[0]
sage: r.apply(b13)
()
```

```
sage: r.apply(b31)
()
```

On a le meme bug (ca se corrige avec de plus grandes extensions a gauche):

```
sage: E = ExtensionType2to1((([a],[b]) for a,b in e), (1,2,3))
sage: E.apply(b23)[0].apply(b13)
(  E(w)   1   2   3
   31             X
   32             X
   3      X   X   X
   13     X   X   X
   23             X
 m(w)=0, ordinary, empty,)
sage: E.apply(b23)[0].apply(b31)
(  E(1w)   1   2   3
    1      X   X   X
    3      X   X   X
   23             X
 m(w)=2, not ord.,
   E(w)   1   2   3
   11     X   X   X
   31     X   X   X
   12             X
   13     X
   23     X
 m(w)=0, not ord., empty)
```

EXAMPLES:

On imagine qu'on vient de faire b12:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:       2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E
  E(w)   1   2   3
   21         X
   31         X
   12     X   X   X
   22     X
   23     X
 m(w)=0, not ord., empty
sage: b12 = WordMorphism({1:[1,2],2:[2],3:[3]})
sage: E.apply(b12)
(  E(2w)   1   2   3
   21          X
   31          X
   12      X   X   X
   22      X
m(w)=0, ordinary,
   E(w)   1   2   3
   21         X
   31         X
   12         X
   22     X   X   X
   23     X
m(w)=0, not ord., empty)

sage: b21 = WordMorphism({1:[1],2:[2,1],3:[3]})
sage: E.apply(b21)
(  E(1w)   1   2   3
   21          X
   12      X   X   X
   13          X
 m(w)=0, ordinary,
   E(w)   1   2   3
```

```
   11        X
   21    X   X   X
   31        X
   12    X
   13    X
 m(w)=0, ordinary, empty)
sage: b23 = WordMorphism({1:[1],2:[2,3],3:[3]})
sage: E.apply(b23)
(  E(3w)   1   2   3
   12     X   X   X
   32     X
   23     X
 m(w)=0, ordinary,
   E(23w)   1   2   3
    31     X   X   X
    23     X
 m(w)=0, ordinary,
   E(w)   1   2   3
   31        X
   12            X
   32            X
   23    X   X   X
   33    X
 m(w)=0, not ord., empty)
```

cardinality()

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:    2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E.cardinality()
5
```

chignons_multiplicity_tuple()

EXAMPLES:

```
sage: L = [(1,3), (2,3), (3,1), (3,2), (3,3)]
sage: E = ExtensionType1to1(L, [1,2,3], ('a', 'b'))
sage: E.chignons_multiplicity_tuple()
('a', 'b', 0)
```

extension_type_1to1()

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:    2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E.extension_type_1to1()
  E(w)   1   2   3
   1         X
   2     X   X   X
   3     X
 m(w)=0, not ord.
```

factors_length_2()

Returns the set of factors of length 2 of the language.

This is computed from the extension type if it was not provided at the construction.

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:    2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E.factors_length_2()
```

```
set([(1, 2), (3, 1), (2, 3), (2, 1), (2, 2)])
```

is_chignons_empty()

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:          2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E.is_chignons_empty()
True
```

is_ordinaire()

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:          2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E.extension_type_1to1()
  E(w)   1   2   3
   1         X
   2     X   X   X
   3     X
 m(w)=0, not ord.
sage: E.is_ordinaire()
False
```

is_valid()

Return whether self is valid, i.e, each left and right extension is non empty.

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:          2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E.is_valid()
True
```

left_extensions()

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:    2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E.left_extensions()
set([1, 2, 3])
```

left_word_extensions()

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:          2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: sorted(E.left_word_extensions())
[word: 12, word: 21, word: 22, word: 23, word: 31]
```

letters_before_and_after()

Returns a pair of dict giving the possible letters that goes before or after a letter.

Computed from the set of factors of length 2 of the language.

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:      2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
```

```
sage: E.factors_length_2()
set([(1, 2), (3, 1), (2, 3), (2, 1), (2, 2)])
sage: E.letters_before_and_after()
(defaultdict(<type 'set'>, {1: set([2, 3]), 2: set([1, 2]), 3: set([2])}),
defaultdict(<type 'set'>, {1: set([2]), 2: set([1, 2, 3]), 3: set([1])}))
```

right_extensions()

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:    2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E.right_extensions()
set([1, 2, 3])
```

right_word_extensions()

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:          2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: sorted(E.right_word_extensions())
[word: 1, word: 2, word: 3]
```

table()

return a table representation of self.

EXAMPLES:

```
sage: L = [((2, 2), (1,)), ((2, 3), (1,)), ((2, 1), (2,)), ((1,
....:          2), (1,)), ((1, 2), (2,)), ((1, 2), (3,)), ((3, 1), (2,))]
sage: E = ExtensionType2to1(L, (1,2,3))
sage: E
  E(w)   1   2   3
   21        X
   31        X
   12    X   X   X
   22    X
   23    X
 m(w)=0, not ord., empty
```

slabbe.bispecial_extension_type.**factors_length_2_from_morphism_and_factors_length_2**(*m*, *F*)

Return the set of factors of lengths two in the image by a morphism of a set of factors of length 2.

INPUT:

- m - endomorphim
- F - set of factors of length 2

EXAMPLES:

```
sage: from slabbe.bispecial_extension_type import factors_length_2_from_morphism_and_factors_length_2
sage: b12 = WordMorphism({1:[1,2],2:[2],3:[3]})
sage: sorted(factors_length_2_from_morphism_and_factors_length_2(b12, []))
[]
sage: sorted(factors_length_2_from_morphism_and_factors_length_2(b12, [(1,1)]))
[(1, 2), (2, 1)]
sage: b23 = WordMorphism({1:[1],2:[2,3],3:[3]})
sage: sorted(factors_length_2_from_morphism_and_factors_length_2(b23, [(1,1)]))
[(1, 1)]
```

slabbe.bispecial_extension_type.**longest_common_prefix**(*L*)

Return the longest common prefix of a list of words.

EXAMPLES:

```
sage: from slabbe.bispecial_extension_type import longest_common_prefix
sage: longest_common_prefix((Word('ab'), Word('abc'), Word('abd')))
word: ab
```

slabbe.bispecial_extension_type.**longest_common_suffix**(*L*)

Return the longest common suffix of a list of words.

EXAMPLES:

```
sage: from slabbe.bispecial_extension_type import longest_common_suffix
sage: longest_common_suffix((Word('abc'), Word('bc'), Word('xabc')))
word: bc
```

CHAPTER

THREE

COMBINATORICS

3.1 Joyal Bijection

Problem suggested by Doron Zeilberger during a talk done at CRM, Montreal, May 11th, 2012 to compare code in different languages. This is a implementation of the Joyal's Bijection using Sage. It will not win for the most brief code, but it is object oriented, documented, reusable, testable and allows introspection.

AUTHOR:

- Sébastien Labbé, May 12th 2012

TODO:

- Base Endofunction class on sage's FiniteSetMap classes (for both element and parent)

EXAMPLES:

3.1.1 Creation of an endofunction

```
sage: L = [7, 0, 6, 1, 4, 7, 2, 1, 5]
sage: f = Endofunction(L)
sage: f
Endofunction:
[0..8] -> [7, 0, 6, 1, 4, 7, 2, 1, 5]
```

3.1.2 Creation of a double rooted tree

```
sage: L = [(0,6),(2,1),(3,1),(4,2),(5,7),(6,4),(7,0),(8,5)]
sage: D = DoubleRootedTree(L, 1, 7)
sage: D
Double rooted tree:
Edges: [(0, 6), (2, 1), (3, 1), (4, 2), (5, 7), (6, 4), (7, 0), (8, 5)]
RootA: 1
RootB: 7
```

3.1.3 Joyal's bijection

From the endofunction `f`, we get a double rooted tree:

```
sage: f.to_double_rooted_tree()
Double rooted tree:
Edges: [(0, 6), (2, 1), (3, 1), (4, 2), (5, 7), (6, 4), (7, 0), (8, 5)]
RootA: 1
RootB: 7
```

From the double rooted tree `D`, we get an endofunction:

```
sage: D.to_endofunction()
Endofunction:
[0..8] -> [7, 0, 6, 1, 4, 7, 2, 1, 5]
```

In fact, we got `D` from `f` and vice versa:

```
sage: D == f.to_double_rooted_tree()
True
sage: f == D.to_endofunction()
True
```

3.1.4 Endofunctions are defined on the set [0, 1, ..., n-1]

As of now, the code supports only endofunctions defined on the set [0, 1, ..., n-1]

```
sage: L = [1, 0, 3, 4, 5, 7, 1]
sage: f = Endofunction(L)
Traceback (most recent call last):
...
ValueError: images of [0..6] must be 0 <= i < 7
```

3.1.5 Another example

From a list `L`, we create an endofunction `f`

```
sage: L = [12, 7, 8, 3, 3, 11, 11, 9, 5, 12, 0, 10, 9]
sage: f = Endofunction(L)
sage: f
Endofunction:
[0..12] -> [12, 7, 8, 3, 3, 11, 11, 9, 5, 12, 0, 10, 9]
```

From `f`, we create a double rooted tree `D`:

```
sage: D = f.to_double_rooted_tree(); D
Double rooted tree:
Edges: [(0, 12), (1, 7), (2, 8), (3, 12), (4, 3), (5, 11),
(6, 11), (7, 9), (8, 5), (10, 0), (11, 10), (12, 9)]
RootA: 9
RootB: 3
```

And from `D`, we create an endofunction:

```
sage: D.to_endofunction()
Endofunction:
[0..12] -> [12, 7, 8, 3, 3, 11, 11, 9, 5, 12, 0, 10, 9]
```

We test that we recover the initial endofunction `f`:

```
sage: f == f.to_double_rooted_tree().to_endofunction()
True
```

3.1.6 A random example

We define the set of all endofunctions on [0..7]:

```
sage: E = Endofunctions(8)
sage: E
Endofunctions of [0..7]
```

We choose a random endofunction on the set [0..7]:

```
sage: f = E.random_element()
sage: f                                # random
Endofunction:
[0..7] -> [5, 5, 0, 4, 5, 0, 1, 1]
```

We construct a double rooted tree from it:

```
sage: f.to_double_rooted_tree()        # random
Double rooted tree:
Edges: [(1, 5), (2, 0), (3, 4), (4, 5), (5, 0), (6, 1), (7, 1)]
RootA: 0
RootB: 5
```

We recover an endofunction from the double rooted tree:

```
sage: f.to_double_rooted_tree().to_endofunction()   # random
Endofunction:
[0..7] -> [5, 5, 0, 4, 5, 0, 1, 1]
```

Finally, we check the bijection:

```
sage: f == f.to_double_rooted_tree().to_endofunction()
True
```

3.1.7 Large random example

```
sage: E = Endofunctions(1000)
sage: f = E.random_element()
sage: f == f.to_double_rooted_tree().to_endofunction()
True
```

TESTS:

We test the limit cases:

```
sage: f = Endofunction([0])
sage: f == f.to_double_rooted_tree().to_endofunction()
True
sage: f = Endofunction([0,1])
sage: f == f.to_double_rooted_tree().to_endofunction()
True
sage: f = Endofunction([1,0])
sage: f == f.to_double_rooted_tree().to_endofunction()
True
```

More extensively:

```
sage: E = Endofunctions(1)
sage: all(f == f.to_double_rooted_tree().to_endofunction() for f in E)
True
sage: E = Endofunctions(2)
```

```
sage: all(f == f.to_double_rooted_tree().to_endofunction() for f in E)
True
sage: E = Endofunctions(3)
sage: all(f == f.to_double_rooted_tree().to_endofunction() for f in E)
True
sage: E = Endofunctions(4)
sage: all(f == f.to_double_rooted_tree().to_endofunction() for f in E)
True
```

TIMING TESTS:

When the extension of the file is .sage:

```
sage: E = Endofunctions(3)
sage: time all(f == f.to_double_rooted_tree().to_endofunction() for f in E) # not tested
True
Time: CPU 0.02 s, Wall: 0.02 s
sage: E = Endofunctions(4)
sage: time all(f == f.to_double_rooted_tree().to_endofunction() for f in E) # not tested
True
Time: CPU 0.22 s, Wall: 0.22 s
sage: E = Endofunctions(5)
sage: time all(f == f.to_double_rooted_tree().to_endofunction() for f in E) # not tested
True
Time: CPU 2.82 s, Wall: 2.82 s
sage: E = Endofunctions(6)
sage: time all(f == f.to_double_rooted_tree().to_endofunction() for f in E) # not tested
True
Time: CPU 45.66 s, Wall: 45.74 s
```

When the extension of the file is .spyx:

```
sage: E = Endofunctions(3)
sage: time all(f == f.to_double_rooted_tree().to_endofunction() for f in E) # not tested
True
Time: CPU 0.02 s, Wall: 0.02 s
sage: E = Endofunctions(4)
sage: time all(f == f.to_double_rooted_tree().to_endofunction() for f in E) # not tested
True
Time: CPU 0.21 s, Wall: 0.21 s
sage: E = Endofunctions(5)
sage: time all(f == f.to_double_rooted_tree().to_endofunction() for f in E) # not tested
True
Time: CPU 2.71 s, Wall: 2.72 s
sage: E = Endofunctions(6)
sage: time all(f == f.to_double_rooted_tree().to_endofunction() for f in E) # not tested
True
Time: CPU 44.08 s, Wall: 44.17 s
```

When the extension of the file is .sage:

```
sage: E = Endofunctions(1000)
sage: f = E.random_element()
sage: time f == f.to_double_rooted_tree().to_endofunction() # not tested
True
Time: CPU 0.09 s, Wall: 0.09 s
sage: E = Endofunctions(10000)
sage: f = E.random_element()
sage: time f == f.to_double_rooted_tree().to_endofunction()  # not tested
True
Time: CPU 2.23 s, Wall: 2.24 s
```

When the extension of the file is .spyx:

```
sage: E = Endofunctions(1000)
sage: f = E.random_element()
sage: time f == f.to_double_rooted_tree().to_endofunction() # not tested
True
Time: CPU 0.11 s, Wall: 0.11 s
sage: E = Endofunctions(10000)
sage: f = E.random_element()
sage: time f == f.to_double_rooted_tree().to_endofunction() # not tested
True
Time: CPU 2.91 s, Wall: 2.93 s
```

class slabbe.joyal_bijection.**DoubleRootedTree**(*edges*, *rootA*, *rootB*)

Bases: object

Returns a double rooted tree.

INPUT:

- edges - list of edges
- rootA - root A
- rootB - root B

EXAMPLES:

```
sage: edges = [(0,5),(1,2),(2,6),(3,2),(4,1),(5,7),(7,1),(8,2),(9,4)]
sage: D = DoubleRootedTree(edges, 6, 0)
sage: D
Double rooted tree:
Edges: [(0, 5), (1, 2), (2, 6), (3, 2), (4, 1), (5, 7), (7, 1), (8, 2), (9, 4)]
RootA: 6
RootB: 0
```

graph()

EXAMPLES:

```
sage: edges = [(0,5),(1,2),(2,6),(3,2),(4,1),(5,7),(7,1),(8,2),(9,4)]
sage: D = DoubleRootedTree(edges, 6, 0)
sage: D.graph()
Graph on 10 vertices
```

skeleton()

EXAMPLES:

```
sage: edges = [(0,5),(1,2),(2,6),(3,2),(4,1),(5,7),(7,1),(8,2),(9,4)]
sage: D = DoubleRootedTree(edges, 6, 0)
sage: D.skeleton()
[0, 5, 7, 1, 2, 6]
```

skeleton_cycles()

EXAMPLES:

```
sage: edges = [(0,5),(1,2),(2,6),(3,2),(4,1),(5,7),(7,1),(8,2),(9,4)]
sage: D = DoubleRootedTree(edges, 6, 0)
sage: D.skeleton()
[0, 5, 7, 1, 2, 6]
sage: D.skeleton_cycles()
[(0,), (1, 5), (2, 7, 6)]
```

to_endofunction()

EXAMPLES:

```
sage: edges = [(0,5),(1,2),(2,6),(3,2),(4,1),(5,7),(7,1),(8,2),(9,4)]
sage: D = DoubleRootedTree(edges, 6, 0)
sage: D.to_endofunction()
```

```
Endofunction:
[0..9] -> [0, 5, 7, 2, 1, 1, 2, 6, 2, 4]
```

TESTS:

```
sage: D = DoubleRootedTree([], 0, 0)
sage: D.to_endofunction()
Endofunction:
[0..0] -> [0]
```

class slabbe.joyal_bijection.**Endofunction**(*L*)

Bases: object

Returns an endofunction.

INPUT:

- L - list of length n containing images of the integers from 0 to n-1 where the images belong to the integers from 0 to n-1.

EXAMPLES:

```
sage: L = [0, 5, 7, 2, 1, 1, 2, 6, 2, 4]
sage: f = Endofunction(L)
sage: f
Endofunction:
[0..9] -> [0, 5, 7, 2, 1, 1, 2, 6, 2, 4]
```

cycle_elements()

Returns the list of all elements in a cycle for this endofunction.

OUTPUT:

list

EXAMPLES:

```
sage: L = [6, 5, 7, 2, 1, 1, 2, 6, 2, 4]
sage: f = Endofunction(L)
sage: f.cycle_elements()   # random order
[0, 6, 7, 2, 1, 5]
```

Note: G.cycle_basis() is not implemented for directed or multiedge graphs (in Networkx). Hence, the cycle_basis method is missing the 2-cycles.

---

skeleton()

Return the skeleton of the endofunction.

OUTPUT:

list

EXAMPLES:

```
sage: L = [0, 5, 7, 2, 1, 1, 2, 6, 2, 4]
sage: f = Endofunction(L)
sage: f.skeleton()
[0, 5, 7, 1, 2, 6]
```

to_double_rooted_tree()

Return the double rooted tree following André Joyal Bijection.

OUTPUT:

Double rooted tree

EXAMPLES:

```
sage: L = [0, 5, 7, 2, 1, 1, 2, 6, 2, 4]
sage: f = Endofunction(L)
sage: f.to_double_rooted_tree()
Double rooted tree:
Edges: [(0, 5), (1, 2), (2, 6), (3, 2), (4, 1), (5, 7), (7, 1), (8, 2), (9, 4)]
RootA: 6
RootB: 0
```

two_cycle_elements()

Iterator over elements in a two-cycle.

EXAMPLES:

```
sage: L = [0, 5, 7, 2, 1, 1, 2, 6, 2, 4]
sage: f = Endofunction(L)
sage: list(f.two_cycle_elements())
[1, 5]
```

class slabbe.joyal_bijection.**Endofunctions**(*n*)

Bases: object

Returns the set of all endofunction on the set [0..n-1].

INPUT:

- n - positive integer

EXAMPLES:

```
sage: Endofunctions(10)
Endofunctions of [0..9]
```

random_element()

Return a random endofunction on [0..n-1].

EXAMPLES:

```
sage: E = Endofunctions(10)
sage: E.random_element()          # random
Endofunction:
[0..9] -> [2, 8, 7, 0, 0, 6, 2, 3, 5, 9]
sage: E.random_element()          # random
Endofunction:
[0..9] -> [8, 7, 7, 5, 4, 1, 0, 3, 8, 6]
```

3.2 Percolation in lattices

This is an implementation of bond percolation. See Chapter 3 of [POG]. See also my blog post Percolation and self-avoiding walks related to this code.

AUTHORS:

- Sebastien Labbe (2012-12-17): initial version, for pog lecture group

REFERENCES:

EXAMPLES:

3.2.1 Bond percolation sample

We construct a bond percolation sample in dimension d=2 with probability of open edges p=0.3:

```
sage: S = BondPercolationSample(p=0.3, d=2)
sage: S
Bond percolation sample d=2 p=0.300
```

An edge is defined uniquely as a starting point in Z^d and an axis direction given by an integer i such that 1 <= i <= d. One may ask if a given edge is in the sample S:

```
sage: ((34,56), 2) in S     # random
False
```

The result is cached so the same answer is returned again:

```
sage: ((34,56), 2) in S     # random
False
sage: ((34,56), 2) in S     # random
False
```

The cluster containing the point zero is returned as an iterator:

```
sage: S.cluster()
<generator object at ...>
```

It may be finite of infinite. If you believe it is finite, you may compute its cardinality. If the cluster is infinite, it will not halt:

```
sage: S.cluster_cardinality() # not tested, might not halt
13
```

For larger values of p, the cluster might be larger if not infinite. In this case you may want to stop the computation at a certain point determined in advance. The following method does this. And it returns the cardinality if it is smaller than the stop value:

```
sage: S = BondPercolationSample(p=0.45, d=2)
sage: S.cluster_cardinality_stop_at(stop=10)          # random
'>=10'
sage: S.cluster_cardinality_stop_at(stop=100)          # random
'>=100'
sage: S.cluster_cardinality_stop_at(stop=1000)          # random
625
```

3.2.2 Bond percolation samples

Construction of 20 bond percolation samples. For each of them, compute the cardinality of the open cluster containing zero:

```
sage: S20 = BondPercolationSamples(p=0.4, d=2, n=20)
sage: S20.cluster_cardinality(stop=100)               # random
[4, 2, 1, 4, 1, 10, 62, 71, 1, 25, 19, 2, 2, 42, '>=100', 1, 18, 2, '>=100', 20]
```

By considering "larger than 100" to be an infinite cluster, this gives a value of 2/20 = 0.10 for the percolation probability:

```
sage: S20.percolation_probability(stop=100)        # random
0.100
```

By increasing the stop value, the computations can be redone again *on the same samples*. In this case, by using a stop value of 1000, we get a value of 0 for the percolation probability of p=0.4:

```
sage: S20.cluster_cardinality(stop=1000)             # random
[4, 2, 1, 4, 1, 10, 62, 71, 1, 25, 19, 2, 2, 42, 176, 1, 18, 2, 186, 20]
sage: S20.percolation_probability(stop=1000)         # random
0.000
```

3.2.3 Percolation probability

One can define the percolation probability function for a given dimension d. It will generate n samples and consider the cluster to be infinite if its cardinality is larger than the given stop value:

```
sage: T = PercolationProbability(d=2, n=10, stop=100)
sage: T
Percolation Probability $\theta(p)$
d = dimension = 2
n = # samples = 10
stop counting at = 100
```

Compute the value for a certain probability p:

```
sage: T(0.4534)             # random
0.300
```

Of course, this value will change for another equal percolation probability since it depends on the samples:

```
sage: T = PercolationProbability(d=2, n=10, stop=100)
sage: T(0.4534)              # random
0.600
```

Anyway, it is usefull to draw the plot of the percolation probability:

```
sage: T = PercolationProbability(d=2, n=10, stop=100)
sage: T.return_plot((0,1), adaptive_recursion=4, plot_points=4)     # optional long
Graphics object consisting of 2 graphics primitives
```

Here we use Sage adaptative recursion algorithm for drawing plots which finds the particular important intervals to ask for more values of the function. See help section of plot function for details. Because T might be long to compute we start with only 4 points

3.2.4 TODO

- Make it 100% doctested (presently 21/24 = 87%)
- Base it on DiscreteSubset code
- Fix tikz2pdf use

Do we want to use?:

```
sage: from sage.sets.set_from_iterator import EnumeratedSetFromIterator
sage: from itertools import count
sage: S = EnumeratedSetFromIterator(count)
sage: S
{0, 1, 2, 3, 4, ...}
```

and ?:

```
sage: M = FiniteSetMaps(["a", "b"], [3, 4, 5]); M
Maps from {'a', 'b'} to {3, 4, 5}
```

3.2.5 Methods and classes

class slabbe.bond_percolation.**BondPercolationSample**(*p*, *d=2*)

Bases: sage.structure.sage_object.SageObject

Let $L^d = (Z^d, E^d)$ be the hypercubic lattice.

A sample contained in the set {0,1}^{E^d}.

An edge e in E is open (=1) in the sample with probability p.

Cached __contains__ method below does the job of memory.

children(*pt*)

Return an iterator over open neighbors of the point pt.

INPUT:

- pt - tuple, point in Z^d
- m - integer, limit

EXAMPLES:

The result is consistent:

```
sage: S = BondPercolationSample(0.5)
sage: list(S.children((0,0)))        # random
[(1, 0), (-1, 0), (0, -1)]
```

Might be different for another sample:

```
sage: S = BondPercolationSample(0.5)
sage: list(S.children((0,0)))        # random
[(-1, 0)]
```

In dimension 3:

```
sage: S = BondPercolationSample(0.5,3)
sage: list(S.children((0,0,0)))        # random
[(1, 0, 0), (0, -1, 0), (0, 0, -1)]
sage: S = BondPercolationSample(0.5,3)
sage: list(S.children((0,0,0)))        # random
[(1, 0, 0), (-1, 0, 0)]
sage: list(S.children((0,0,0)))        # random
[(1, 0, 0), (-1, 0, 0)]

sage: S = BondPercolationSample(1,2)
sage: list(S.children((0,0)))        # random
[(1, 0), (-1, 0), (0, 1), (0, -1)]
sage: S = BondPercolationSample(1,3)
sage: list(S.children((0,0,0)))        # random
[(1, 0, 0), (-1, 0, 0), (0, 1, 0), (0, -1, 0), (0, 0, 1), (0, 0, -1)]
```

cluster(*pt=None*)

Return an iterator over the open cluster containing the point pt.

INPUT:

- pt - tuple, point in Z^d. If None, pt=zero is considered.

EXAMPLES:

sage: S = BondPercolationSample(0.5) sage: it = S.cluster() sage: next(it) (0, 0)

cluster_cardinality(*pt=None*)

INPUT:

- `pt` - tuple, point in Z^d. If None, pt=zero is considered.

EXAMPLES:

```
sage: BondPercolationSample(0.01).cluster_cardinality() # random
1
sage: BondPercolationSample(0.4).cluster_cardinality()  # random
28
```

cluster_cardinality_stop_at(*stop*, *pt=None*)

Return the cardinality of the cluster or the strin ">=STOP" if the size is larger than stop value.

INPUT:

- `stop` - integer
- `pt` - tuple, point in Z^d. If None, pt=zero is considered.

EXAMPLES:

```
sage: S = BondPercolationSample(0.3,2)
sage: S.cluster_cardinality()                # random
13
sage: S.cluster_cardinality_stop_at(1000)    # random
13
sage: S.cluster_cardinality_stop_at(100)     # random
13
sage: S.cluster_cardinality_stop_at(10)      # random
'>=10'
```

cluster_in_box(*m*, *pt=None*)

Return the cluster (as a list) in the primal box [-m,m]^d containing the point pt.

INPUT:

- `m` - integer
- `pt` - tuple, point in Z^d. If None, pt=zero is considered.

EXAMPLES:

```
sage: S = BondPercolationSample(0.3,2)
sage: S.cluster_in_box(2)         # random
[(-2, -2), (-2, -1), (-1, -2), (-1, -1), (-1, 0), (0, 0)]
```

edges_in_box(*m*)

Return an iterator over all edges in the primal box [-m,m]^d.

INPUT:

- `m` - integer

EXAMPLES:

```
sage: S = BondPercolationSample(1,2)
sage: for a in S.edges_in_box(1): print a
((-1, -1), (0, -1))
((-1, -1), (-1, 0))
((-1, 0), (0, 0))
((-1, 0), (-1, 1))
((0, -1), (1, -1))
```

```
((0, -1), (0, 0))
((0, 0), (1, 0))
((0, 0), (0, 1))
```

neighbor(*pt*, *d*)

Return the neighbors of the point pt in direction d.

INPUT:

- `pt` - tuple, point in Z^d
- `direction` - integer, possible values are 1, 2, ..., d and -1, -2, ..., -d.

EXAMPLES:

sage: S = BondPercolationSample(0.5,2) sage: S.neighbor((2,3),1) (3, 3) sage: S.neighbor((2,3),2) (2, 4) sage: S.neighbor((2,3),-1) (1, 3) sage: S.neighbor((2,3),-2) (2, 2)

plot(*m*, *pointsize=100*, *thickness=3*, *axes=False*)

Return 2d graphics object contained in the primal box [-m,m]^d.

INPUT:

- `pointsize`, integer (default:`100`),
- `thickness`, integer (default:3),
- `axes`, bool (default:`False`),

EXAMPLES:

```
sage: S = BondPercolationSample(0.5,2)
sage: S.plot(2)           # optional long
```

It works in 3d!!:

```
sage: S = BondPercolationSample(0.5,3)
sage: S.plot(3, pointsize=10, thickness=1)     # optional long
Graphics3d Object
```

save_pdf(*m*)

EXAMPLES:

```
sage: d = 2
sage: BondPercolationSample(0.3, d).save_pdf(20)    # optional long
Creation du fichier tikz_sample_d2_p300_m20.tikz
Using template '/Users/slabbe/.tikz2pdf.tex'.
tikz2pdf: calling pdflatex...
tikz2pdf: Output written to 'tikz_sample_d2_p300_m20.pdf'.
```

tikz(*m*)

Return tikz code.

EXAMPLES:

```
sage: S = BondPercolationSample(0.5,2)
sage: S.tikz(2)
\begin{tikzpicture}
[inner sep=0pt,thick,
 reddot/.style={fill=red,draw=red,circle,minimum
 size=5pt}]
\clip (-2.4, -2.4) rectangle (2.4, 2.4);
\draw (..., ...) -- (..., ...);
...
\node[reddot] at (..., ...) {};
```

```
\node[circle,fill=none,draw=red,minimum size=0.8cm,ultra thick,inner sep=0pt] at (0,0) {};
\node[above right] at (0,0) {$(0, 0)$};
\end{tikzpicture}
```

zero()

EXAMPLES:

```
sage: S = BondPercolationSample(0.5,3)
sage: S.zero()
(0, 0, 0)

sage: S = BondPercolationSample(0.5,5)
sage: S.zero()
(0, 0, 0, 0, 0)
```

class slabbe.bond_percolation.**BondPercolationSamples**(*p*, *d*, *n*)

Bases: sage.structure.sage_object.SageObject

Return a list of n BondPercolationSample of given parameter p and dimension d.

EXAMPLES:

```
sage: BondPercolationSamples(0.2,2,3)
<class 'slabbe.bond_percolation.BondPercolationSamples'>
```

cluster_cardinality(*stop*)

Return the list of cardinality of the cluster for each sample or +Infinity if the size is larger than stop value.

INPUT:

- stop - integer

EXAMPLES:

```
sage: S20 = BondPercolationSamples(p=0.2, d=2, n=20)
sage: S20.cluster_cardinality(100)       # random
[1, 4, 1, 2, 2, 6, 5, 1, 5, 9, 2, 2, 2, 1, 2, 4, 4, 3, 2, 1]

sage: d = 2
sage: n = 5
sage: for p in srange(0,1,0.1): print p,BondPercolationSamples(p,d,n).cluster_cardinality(100) # optional long
0.000000000000000 [1, 1, 1, 1, 1]
0.100000000000000 [5, 1, 2, 1, 2]
0.200000000000000 [3, 1, 4, 1, 1]
0.300000000000000 [11, 7, 2, 1, 3]
0.400000000000000 [3, 3, 35, 10, '>=100']
0.500000000000000 ['>=100', '>=100', '>=100', '>=100', 26]
0.600000000000000 ['>=100', '>=100', '>=100', '>=100', '>=100']
0.700000000000000 ['>=100', '>=100', '>=100', '>=100', '>=100']
0.800000000000000 ['>=100', '>=100', '>=100', '>=100', '>=100']
0.900000000000000 ['>=100', '>=100', '>=100', '>=100', '>=100']
```

ntimes_over_size(*stop*)

EXAMPLES:

```
sage: S = BondPercolationSamples(0.2,2,20)
sage: S.ntimes_over_size(100)    # random
0
sage: S = BondPercolationSamples(0.4,2,20)
sage: S.ntimes_over_size(100)    # random
1
sage: S = BondPercolationSamples(0.5,2,20)
sage: S.ntimes_over_size(100)    # random
17
```

percolation_probability(*stop*)
x.__init__(...) initializes x; see help(type(x)) for signature

class slabbe.bond_percolation.**PercolationProbability**(*d*, *n*, *stop*, *verbose=False*)
Bases: sage.structure.sage_object.SageObject

EXAMPLES:

```
sage: f = PercolationProbability(d=2, n=10, stop=100)
sage: f
Percolation Probability $\theta(p)$
d = dimension = 2
n = # samples = 10
stop counting at = 100
```

return_plot(*interval=(0, 1)*, *adaptive_recursion=4*, *plot_points=4*, *adaptive_tolerance=0.1*)
Return a plot of percolation probability using basic sage plot settings.

INPUT:

- interval, default=(0,1)
- adaptive_recursion, default=0
- plot_points, default=10
- adaptive_tolerance default=0.10

EXAMPLES:

```
sage: T = PercolationProbability(d=2, n=10, stop=100)
sage: T.return_plot()           # optional long
Graphics object consisting of 1 graphics primitive
```

slabbe.bond_percolation.**compute_percolation_probability**(*range_p*, *d*, *n*, *stop*)
EXAMPLES:

```
sage: from slabbe.bond_percolation import compute_percolation_probability
sage: compute_percolation_probability(srange(0,0.8,0.1), d=2, n=5, stop=100) # random
d = 2, n = number of samples = 5
stop counting at = 100
p=0.0000, Theta=0.000, if |C|< 100 then max|C|=1
p=0.1000, Theta=0.000, if |C|< 100 then max|C|=1
p=0.2000, Theta=0.000, if |C|< 100 then max|C|=5
p=0.3000, Theta=0.000, if |C|< 100 then max|C|=6
p=0.4000, Theta=0.000, if |C|< 100 then max|C|=31
p=0.5000, Theta=1.00, if |C|< 100 then max|C|=-Infinity
p=0.6000, Theta=1.00, if |C|< 100 then max|C|=-Infinity
p=0.7000, Theta=1.00, if |C|< 100 then max|C|=-Infinity
```

```
sage: range_p = srange(0,0.8,0.1)
sage: compute_percolation_probability(range_p, d=2, n=5, stop=100) # not tested
d = 2, n = number of samples = 5
stop counting at = 100
p=0.0000, Theta=0.000, if |C|< 100 then max|C|=1
p=0.1000, Theta=0.000, if |C|< 100 then max|C|=1
p=0.2000, Theta=0.000, if |C|< 100 then max|C|=5
p=0.3000, Theta=0.000, if |C|< 100 then max|C|=6
p=0.4000, Theta=0.000, if |C|< 100 then max|C|=31
p=0.5000, Theta=1.00, if |C|< 100 then max|C|=-Infinity
p=0.6000, Theta=1.00, if |C|< 100 then max|C|=-Infinity
p=0.7000, Theta=1.00, if |C|< 100 then max|C|=-Infinity
```

```
sage: range_p = srange(0.45,0.55,0.01)
sage: compute_percolation_probability(range_p, d=2, n=10, stop=1000) # not tested
d = 2, n = number of samples = 10
stop counting at = 1000
```

```
p=0.4500, Theta=0.000, if |C|< 1000 then max|C|=378
p=0.4600, Theta=0.000, if |C|< 1000 then max|C|=475
p=0.4700, Theta=0.000, if |C|< 1000 then max|C|=514
p=0.4800, Theta=0.100, if |C|< 1000 then max|C|=655
p=0.4900, Theta=0.700, if |C|< 1000 then max|C|=274
p=0.5000, Theta=0.700, if |C|< 1000 then max|C|=975
p=0.5100, Theta=0.700, if |C|< 1000 then max|C|=16
p=0.5200, Theta=0.700, if |C|< 1000 then max|C|=125
p=0.5300, Theta=0.900, if |C|< 1000 then max|C|=4
p=0.5400, Theta=0.700, if |C|< 1000 then max|C|=6
```

```
sage: range_p = srange(0.475,0.485,0.001)
sage: compute_percolation_probability(range_p, d=2, n=10, stop=1000) # not tested
d = 2, n = number of samples = 10
stop counting at = 1000
p=0.4750, Theta=0.200, if |C|< 1000 then max|C|=718
p=0.4760, Theta=0.200, if |C|< 1000 then max|C|=844
p=0.4770, Theta=0.200, if |C|< 1000 then max|C|=566
p=0.4780, Theta=0.500, if |C|< 1000 then max|C|=257
p=0.4790, Theta=0.200, if |C|< 1000 then max|C|=566
p=0.4800, Theta=0.300, if |C|< 1000 then max|C|=544
p=0.4810, Theta=0.300, if |C|< 1000 then max|C|=778
p=0.4820, Theta=0.500, if |C|< 1000 then max|C|=983
p=0.4830, Theta=0.300, if |C|< 1000 then max|C|=473
p=0.4840, Theta=0.500, if |C|< 1000 then max|C|=411
```

```
sage: range_p = srange(0.47,0.48,0.001)
sage: compute_percolation_probability(range_p, d=2, n=20, stop=2000)  # not tested
d = 2, n = number of samples = 20
stop counting at = 2000
p=0.4700, Theta=0.0500, if |C|< 2000 then max|C|=1666
p=0.4710, Theta=0.100, if |C|< 2000 then max|C|=1665
p=0.4720, Theta=0.000, if |C|< 2000 then max|C|=1798
p=0.4730, Theta=0.0500, if |C|< 2000 then max|C|=1717
p=0.4740, Theta=0.150, if |C|< 2000 then max|C|=1924
p=0.4750, Theta=0.150, if |C|< 2000 then max|C|=1893
p=0.4760, Theta=0.150, if |C|< 2000 then max|C|=1458
p=0.4770, Theta=0.150, if |C|< 2000 then max|C|=1573
p=0.4780, Theta=0.200, if |C|< 2000 then max|C|=1762
p=0.4790, Theta=0.250, if |C|< 2000 then max|C|=951
```

slabbe.bond_percolation.**percolation_graphics_array**(*range_p*, *d*, *m*, *ncols=3*)

EXAMPLES:

```
sage: from slabbe.bond_percolation import percolation_graphics_array
sage: percolation_graphics_array(srange(0.1,1,0.1), d=2, m=5)    # optional long
sage: P = percolation_graphics_array(srange(0.45,0.55,0.01), d=2, m=5)  # optional long
sage: P.save('array_p45_p55_m5.png')     # not tested
sage: P = percolation_graphics_array(srange(0.45,0.55,0.01), d=2, m=10) # optional long
sage: P.save('array_p45_p55_m10.png')    # not tested
```

CHAPTER

FOUR

PYTHON CLASS INHERITANCE

4.1 Fruit

This file is an example of Sage conventions for syntax and documentation. It also serves as an example to explain class inheritance: methods defined for fruits are automatically defined for bananas and strawberries.

AUTHORS:

- Sébastien Labbé, 2010-2013

EXAMPLES:

```
sage: f = Fruit(5)
sage: f
A fruit of 5 kilos.
sage: f.weight()
5
sage: f.is_a_fruit()
True
```

Because of class inheritance which says that a banana is a fruit and a strawberry is a fruit, the methods written for fruits are defined for bananas and strawberries automatically:

```
sage: s = Strawberry(32)
sage: s
A strawberry of 32 kilos.
sage: s.weight()
32
sage: s.is_a_fruit()
True


sage: b = Banana(13)
sage: b
A banana of 13 kilos.
sage: b.weight()
13
sage: b.is_a_fruit()
True


sage: s = Strawberry(32)
sage: t = Strawberry(7)
sage: s
A strawberry of 32 kilos.
sage: t
A strawberry of 7 kilos.
sage: s + t
A strawberry of 1073 kilos.
```

class slabbe.fruit.**Banana**(*weight=1*)

Bases: slabbe.fruit.Fruit

Creates a banana.

INPUT:

- weight - number, in kilos

OUTPUT:

Banana

EXAMPLES:

```
sage: b = Banana(9)
sage: b
A banana of 9 kilos.
```

TESTS:

Testing that pickle works:

```
sage: loads(dumps(b))
A banana of 9 kilos.
sage: b == loads(dumps(b))
True
```

Running the test suite:

```
sage: TestSuite(b).run()
```

class slabbe.fruit.**Fruit**(*weight=1*)

Bases: sage.structure.sage_object.SageObject

Creates a fruit.

INPUT:

- weight - number, in kilos

OUTPUT:

Fruit

EXAMPLES:

```
sage: f = Fruit(5)
sage: f
A fruit of 5 kilos.
```

is_a_fruit()

Returns True if it is a fruit.

OUTPUT:

Boolean

EXAMPLES:

```
sage: f = Fruit(3)
sage: f.is_a_fruit()
True
```

weight()

Return the weight.

OUTPUT:

Number

EXAMPLES:

```
sage: f = Fruit(3)
sage: f.weight()
3
```

class slabbe.fruit.**Strawberry**(*weight=1*)

Bases: slabbe.fruit.Fruit

Creates a strawberry.

INPUT:

- `weight` - number, in kilos

OUTPUT:

Strawberry

EXAMPLES:

```
sage: s = Strawberry(34)
sage: s
A strawberry of 34 kilos.
```

TESTS:

Testing that pickle works:

```
sage: loads(dumps(s))
A strawberry of 34 kilos.
sage: s == loads(dumps(s))
True
```

Running the test suite:

```
sage: TestSuite(s).run()
```

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

[LR2014] Labbé, Sébastien, and Christophe Reutenauer. A d-dimensional Extension of Christoffel Words. arXiv:1404.4021 (April 15, 2014).

[WVL1984] Wijshoff, H. A. G, et J. Van Leeuwen. Arbitrary versus periodic storage schemes and tessellations of the plane using one type of polyomino. INFORM. AND CONTROL 62 (1984): 1-25.

[BN1991] Beauquier, D., and M. Nivat. On translating one polyomino to tile the plane. Discrete & Computational Geometry 6 (1991): 575-592. doi:10.1007/BF02574705

[BFP2009] S. Brlek, J.-M Fédou, X. Provençal, On the Tiling by Translation Problem, Discrete Applied Mathematics 157 Issue 3 (2009) 464-475. doi:10.1016/j.dam.2008.05.026

[BBL2012] A. Blondin Massé, S. Brlek, S. Labbé, A parallelogram tile fills the plane by translation in at most two distinct ways, Discrete Applied Mathematics 160 (2012) 1011-1018. doi:10.1016/j.dam.2011.12.023

[BBGL2011] A. Blondin Massé, S. Brlek, A. Garon, S. Labbé, Two infinite families of polyominoes that tile the plane by translation in two distinct ways, Theoret. Comput. Sci. 412 (2011) 4778-4786. doi:10.1016/j.tcs.2010.12.034

[BGL2012] A. Blondin Massé, A. Garon, S. Labbé, Combinatorial properties of double square tiles, Theoretical Computer Science, Available online 2 November 2012. doi:10.1016/j.tcs.2012.10.040

[K65] William Kolakoski, proposal 5304, American Mathematical Monthly 72 (1965), 674; for a partial solution, see "Self Generating Runs," by Necdet Üçoluk, Amer. Math. Mon. 73 (1966), 681-2.

[O39] R. Oldenburger, Exponent trajectories in dynamical systems, Trans. Amer. Math. Soc. 46 (1939), 453–466.

[BL2014] V. Berthé, S. Labbé, Factor Complexity of S-adic sequences generated by the Arnoux-Rauzy-Poincaré Algorithm. arXiv:1404.4189 (April, 2014).

[POG] Geoffrey Grimmett, Probability on Graphs, http://www.statslab.cam.ac.uk/~grg/books/pgs.html

PYTHON MODULE INDEX

S

INDEX

D

E

F

G

H

I

K

L

M

N

P

R

S

T

U

V

W

Z