Sage Reference Manual: Knot Theory

Release 7.3

The Sage Development Team

Aug 05, 2016

CONTENTS

CHAPTER

ONE

KNOTS

AUTHORS:

- Miguel Angel Marco Buzunariz
- Amit Jamadagni

class `sage.knots.knot.`**`Knot`**`(`*data*, *check=True*`)`

Bases: *`sage.knots.link.Link`*

A knot.

A knot is defined as embedding of the circle $\mathbb{S}^1$ in the 3-dimensional sphere $\mathbb{S}^3$, considered up to ambient isotopy. They represent the physical idea of a knotted rope, but with the particularity that the rope is closed. That is, the ends of the rope are joined.

See also:

`Link`

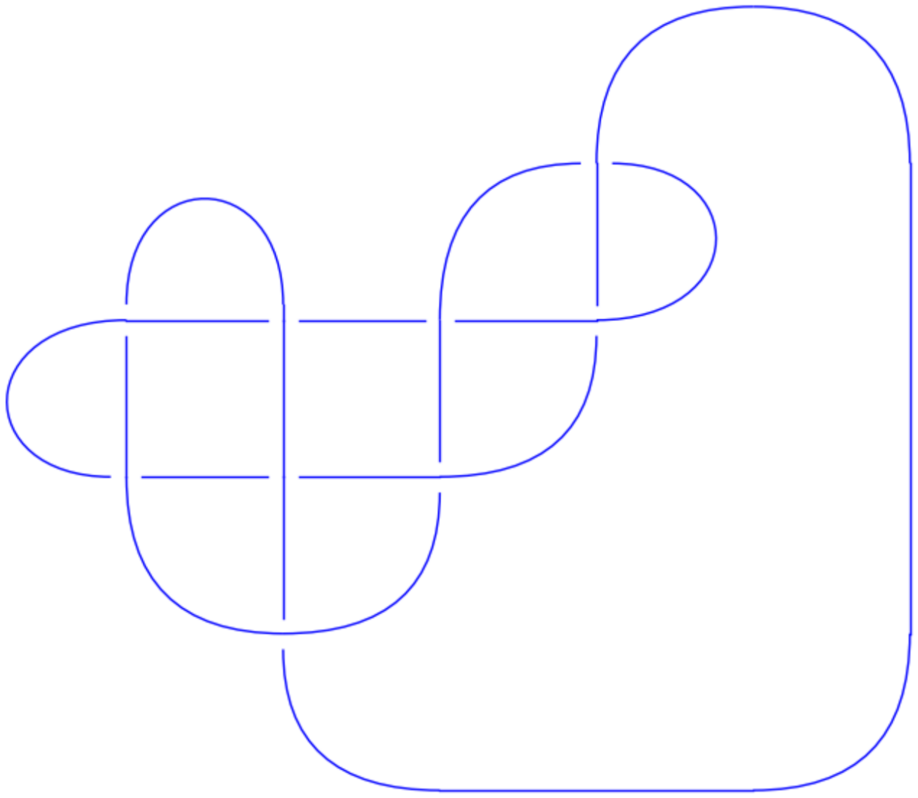
INPUT:

- `data` – see *`Link`* for the allowable inputs
- `check` – optional, default `True`. If `True`, make sure that the data define a knot, not a link

EXAMPLES:

We construct the knot 8_{14} and compute some invariants:

```
sage: B = BraidGroup(4)
sage: K = Knot(B([1,1,1,2,-1,2,-3,2,-3]))
```



```
sage: K.alexander_polynomial()
-2*t^-2 + 8*t^-1 - 11 + 8*t - 2*t^2
sage: K.jones_polynomial()
t^7 - 3*t^6 + 4*t^5 - 5*t^4 + 6*t^3 - 5*t^2 + 4*t + 1/t - 2
sage: K.determinant()
31
sage: K.signature()
-2
```

REFERENCES:

- Wikipedia article Knot_(mathematics)

Todo

- Implement the connect sum of two knots.
- Make a class Knots for the monoid of all knots and have this be an element in that monoid.

arf_invariant ( )

Return the Arf invariant.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: K = Knot(B([-1, 2, 1, 2]))
sage: K.arf_invariant()
```

```
0
sage: B = BraidGroup(8)
sage: K = Knot(B([-2, 3, 1, 2, 1, 4]))
sage: K.arf_invariant()
0
sage: K = Knot(B([1, 2, 1, 2]))
sage: K.arf_invariant()
1
```

dt_code ( )

Return the DT code of `self`.

ALGORITHM:

The DT code is generated by the following way:

Start moving along the knot, as we encounter the crossings we start numbering them, so every crossing has two numbers assigned to it once we have traced the entire knot. Now we take the even number associated with every crossing.

The following sign convention is to be followed:

Take the even number with a negative sign if it is an overcrossing that we are encountering.

OUTPUT: DT code representation of the knot

EXAMPLES:

```
sage: K = Knot([[1,5,2,4],[5,3,6,2],[3,1,4,6]])
sage: K.dt_code()
[4, 6, 2]
sage: B = BraidGroup(4)
sage: K = Knot(B([1, 2, 1, 2]))
sage: K.dt_code()
[4, -6, 8, -2]
sage: K = Knot([[[1, -2, 3, -4, 5, -1, 2, -3, 4, -5]], [1, 1, 1, 1, 1]])
sage: K.dt_code()
[6, 8, 10, 2, 4]
```

CHAPTER

TWO

LINKS

A knot is defined as embedding of the circle $\mathbb{S}^1$ in the 3-dimensional sphere $\mathbb{S}^3$, considered up to ambient isotopy. They represent the physical idea of a knotted rope, but with the particularity that the rope is closed. That is, the ends of the rope are joined.

A link is an embedding of one or more copies of $\mathbb{S}^1$ in $\mathbb{S}^3$, considered up to ambient isotopy. That is, a link represents the idea of one or more tied ropes. Every knot is a link, but not every link is a knot.

Generically, the projection of a link on $\mathbf{R}^2$ is a curve with crossings. The crossings are represented to show which strand goes over the other. This curve is called a planar diagram of the link. If we remove the crossings, the resulting connected components are segments. These segments are called the edges of the diagram.

REFERENCES:

- Wikipedia article Knot_(mathematics)

See also:

There are also tables of link and knot invariants at http://www.indiana.edu/~knotinfo/ and http://www.indiana.edu/~linkinfo/.

AUTHORS:

- Miguel Angel Marco Buzunariz
- Amit Jamadagni

class `sage.knots.link.`**`Link`**`(`*data*`)`

Bases: `object`

A link.

A link is an embedding of one or more copies of $\mathbb{S}^1$ in $\mathbb{S}^3$, considered up to ambient isotopy. That is, a link represents the idea of one or more tied ropes. Every knot is a link, but not every link is a knot.

A link can be created by using one of the conventions mentioned below:

Braid:

•The closure of a braid is a link:

```
sage: B = BraidGroup(8)
sage: L = Link(B([-1, -1, -1, -2, 1, -2, 3, -2, 3]))
sage: L
Link with 1 component represented by 9 crossings
sage: L = Link(B([1, 2, 1, -2, -1]))
sage: L
Link with 2 components represented by 5 crossings
```

Note: The strands of the braid that have no crossings at all are removed.

- Oriented Gauss Code:

  Label the crossings from 1 to n (where n is the number of crossings) and start moving along the link. Trace every component of the link, by starting at a particular point on one component of the link and writing down each of the crossings that you encounter until returning to the starting point. The crossings are written with sign depending on whether we cross them as over or undercrossing. Each component is then represented as a list whose elements are the crossing numbers. A second list of $+1$ and -1's keeps track of the orientation of each crossing:

```
sage: L = Link([[[-1, 2, 3, -4, 5, -6, 7, 8, -2, -5, 6, 1, -8, -3, 4, -7]],
....:           [-1, -1, -1, -1, 1, 1, -1, 1]])
sage: L
Link with 1 component represented by 8 crossings
```

  For links there may be more than one component and the input is as follows:

```
sage: L = Link([[[-1, 2], [-3, 4], [1, 3, -4, -2]], [-1, -1, 1, 1]])
sage: L
Link with 3 components represented by 4 crossings
```

- Planar Diagram (PD) Code:

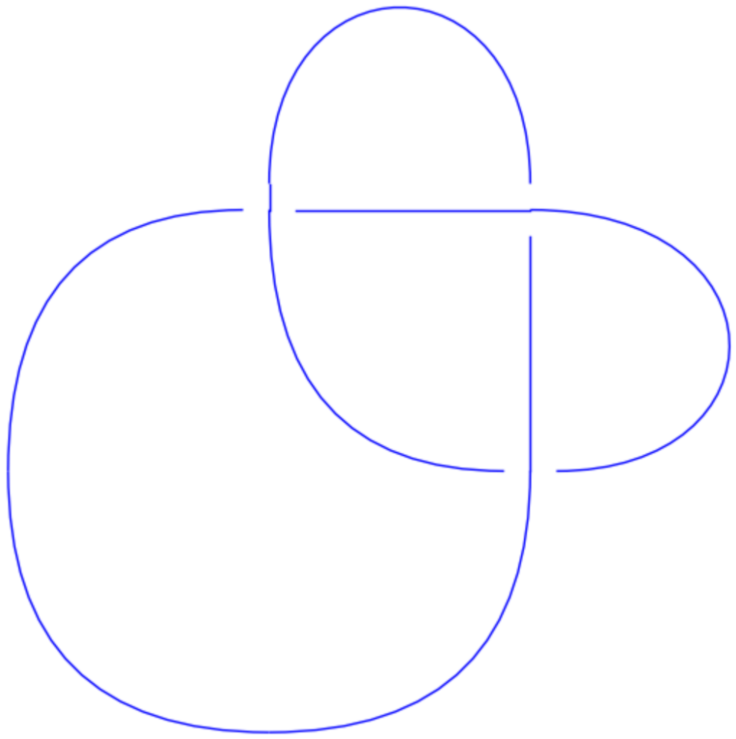
  The diagram of the link is formed by segments that are adjacent to the crossings. Label each one of this segments with a positive number, and for each crossing, write down the four incident segments. The order of these segments is clockwise, starting with the incoming undercrossing.

  There is no particular distinction between knots and links for this input.

EXAMPLES:

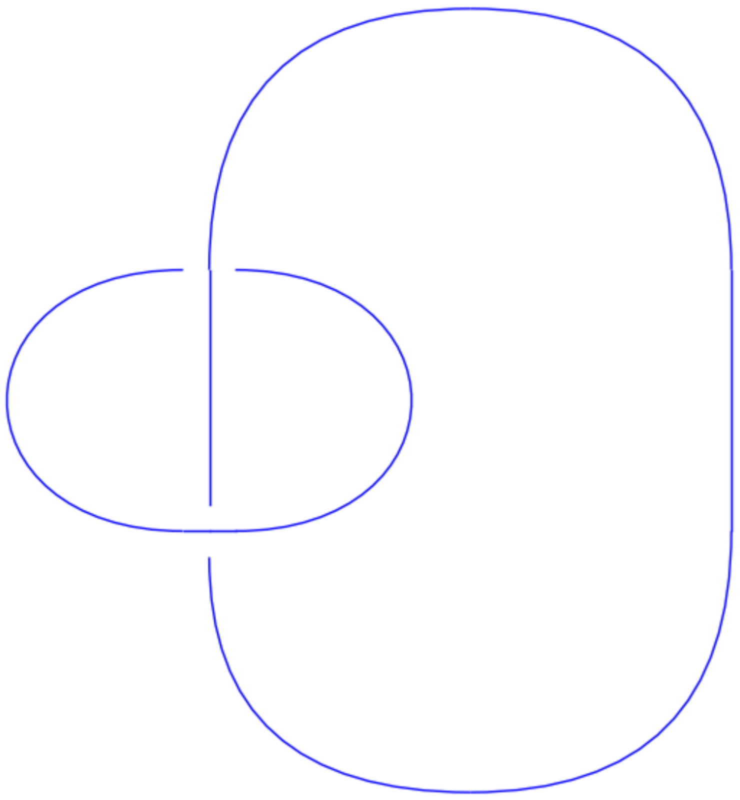
One of the representations of the trefoil knot:

```
sage: L = Link([[1, 5, 2, 4], [5, 3, 6, 2], [3, 1, 4, 6]])
sage: L
Link with 1 component represented by 3 crossings
```



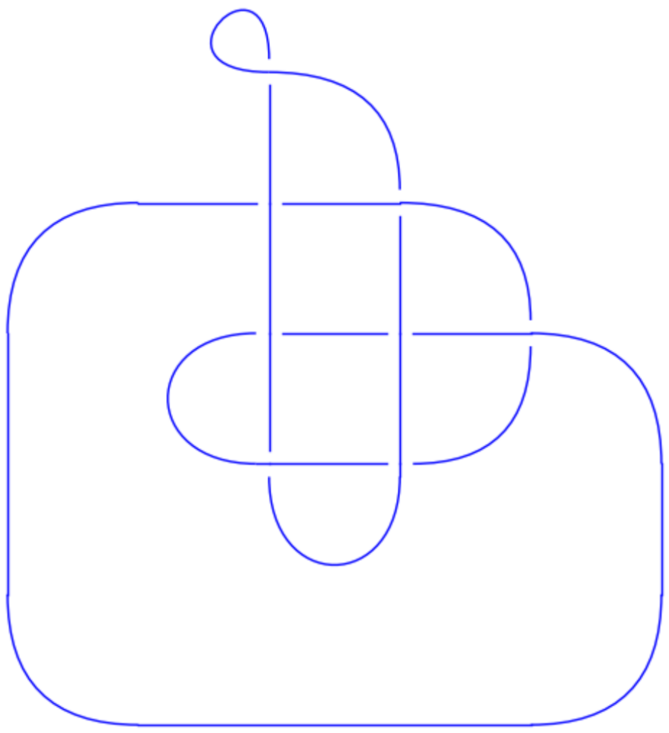
One of the representations of the Hopf link:

```
sage: L = Link([[1, 4, 2, 3], [4, 1, 3, 2]])
sage: L
Link with 2 components represented by 2 crossings
```

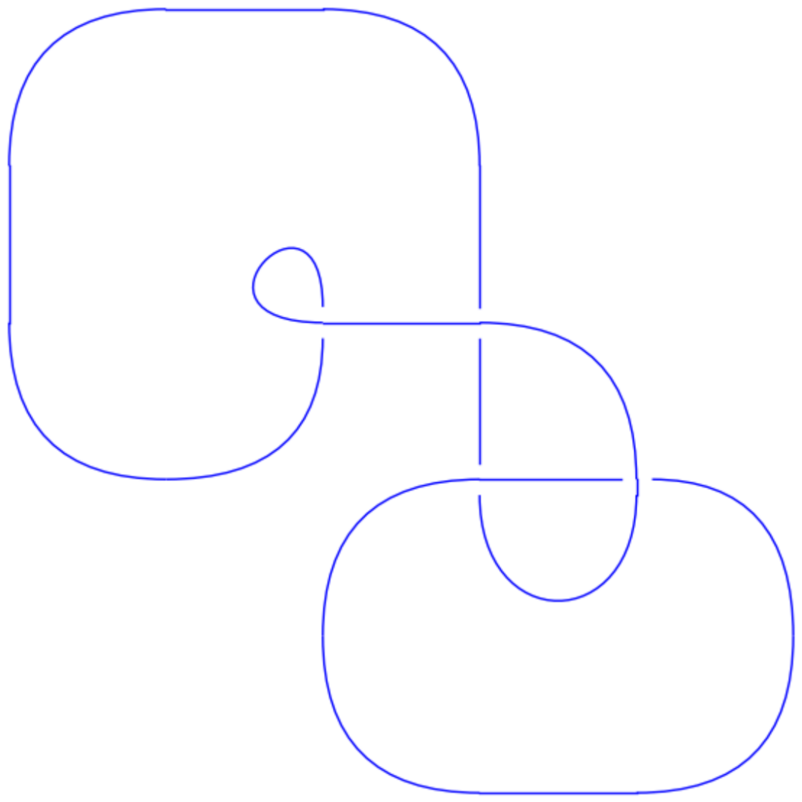


We can construct links from from the braid group:

```
sage: B = BraidGroup(4)
sage: L = Link(B([-1, -1, -1, -2, 1, -2, 3, -2]))
sage: L
Link with 2 components represented by 8 crossings
```

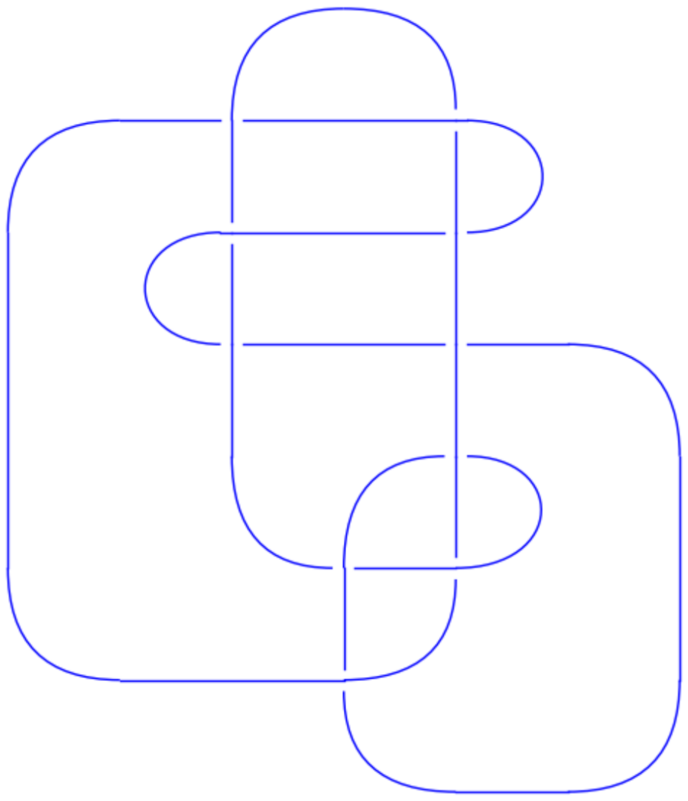


```
sage: L = Link(B([1, 2, 1, 3]))
sage: L
Link with 2 components represented by 4 crossings
```



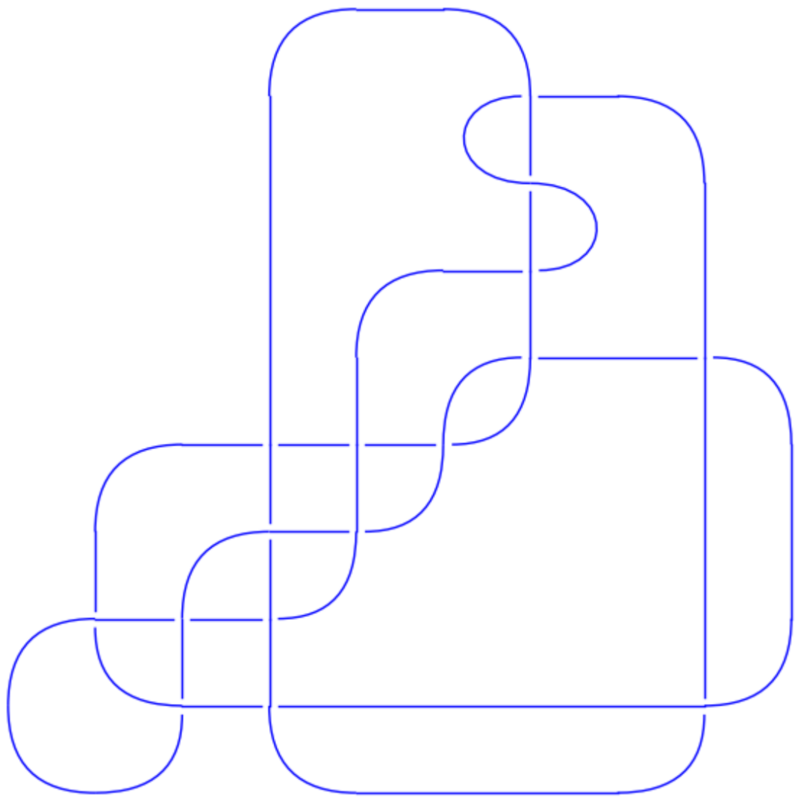
We construct the "monster" unknot using a planar code, and then construct the oriented Gauss code and braid representation:

```
sage: L = Link([[3,1,2,4], [8,9,1,7], [5,6,7,3], [4,18,6,5],
....:           [17,19,8,18], [9,10,11,14], [10,12,13,11],
....:           [12,19,15,13], [20,16,14,15], [16,20,17,2]])
sage: L.oriented_gauss_code()
[[[1, -4, 3, -1, 10, -9, 6, -7, 8, 5, 4, -3, 2, -6, 7, -8, 9, -10, -5, -2]],
 [1, -1, 1, 1, 1, -1, -1, -1, -1, -1]]
sage: L.braid()
s0*s1^-1*s2^-1*s3^-1*s2*s1^-1*s0^-1*s1*s2^2*s1^-1*s3*s2*s1^-3
```



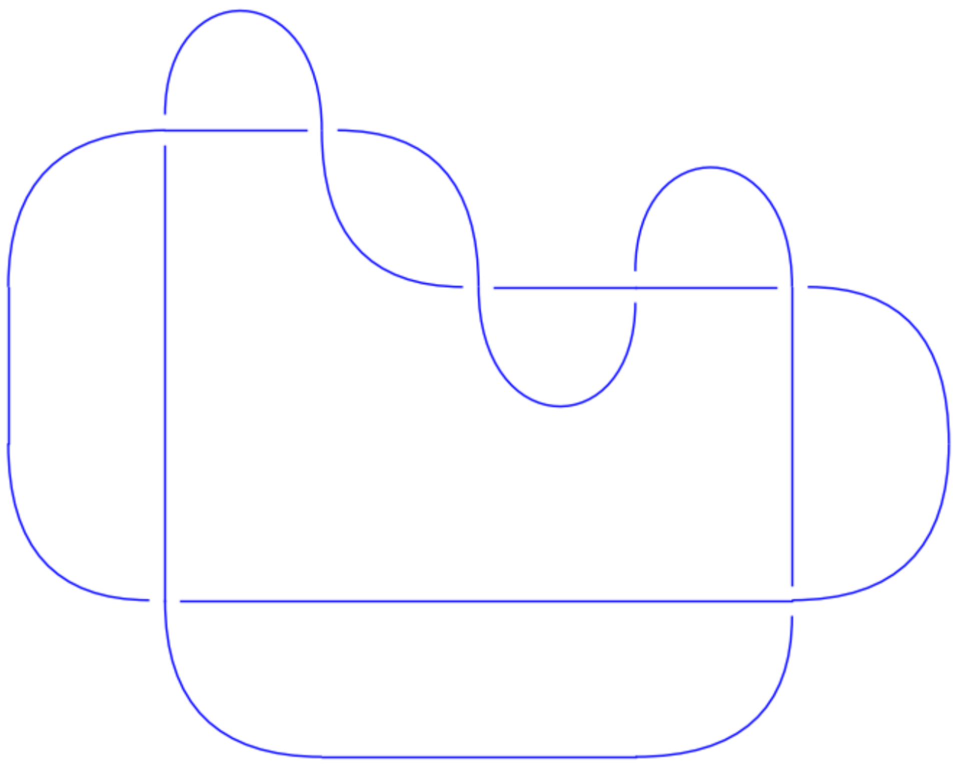
We construct the Ochiai unknot by using an oriented Gauss code:

```
sage: L = Link([[[1,-2,-3,-8,-12,13,-14,15,-7,-1,2,-4,10,11,-13,12,
....:             -11,-16,4,3,-5,6,-9,7,-15,14,16,-10,8,9,-6,5]],
....:            [-1,-1,1,1,1,1,-1,1,1,-1,1,-1,-1,-1,-1,-1]])
sage: L.pd_code()
[[10, 2, 11, 1], [2, 12, 3, 11], [3, 20, 4, 21], [12, 19, 13, 20],
 [21, 32, 22, 1], [31, 22, 32, 23], [9, 25, 10, 24], [4, 29, 5, 30],
 [23, 30, 24, 31], [28, 14, 29, 13], [17, 14, 18, 15], [5, 17, 6, 16],
 [15, 7, 16, 6], [7, 27, 8, 26], [25, 9, 26, 8], [18, 28, 19, 27]]
```



We construct the knot 7_1 and compute some invariants:

```
sage: B = BraidGroup(2)
sage: L = Link(B([1]*7))
```



```
sage: L.alexander_polynomial()
t^-3 - t^-2 + t^-1 - 1 + t - t^2 + t^3
sage: L.jones_polynomial()
-t^10 + t^9 - t^8 + t^7 - t^6 + t^5 + t^3
sage: L.determinant()
7
sage: L.signature()
-6
```

The links here have removed components in which no strand is used:

```
sage: B = BraidGroup(8)
sage: b = B([1])
sage: L = Link(b)
sage: b.components_in_closure()
7
sage: L.number_of_components()
1
sage: L.braid().components_in_closure()
1
sage: L.braid().parent()
Braid group on 2 strands
```

Warning: Equality of knots is done by comparing the corresponding braids, which may give false negatives.

Note: The behavior of removing unused strands from an element of a braid group may change without notice in the future. Do not rely on this feature.

Todo

Implement methods to creating new links from previously created links.

alexander_polynomial (*var='t'*)

Return the Alexander polynomial of `self` .

INPUT:

- `var` – (default: `'t'`) the variable in the polynomial

EXAMPLES:

We begin by computing the Alexander polynomial for the figure-eight knot:

```
sage: B = BraidGroup(3)
sage: L = Link(B([1, -2, 1, -2]))
sage: L.alexander_polynomial()
-t^-1 + 3 - t
```

The "monster" unknot:

```
sage: L = Link([[3,1,2,4],[8,9,1,7],[5,6,7,3],[4,18,6,5],
....:           [17,19,8,18],[9,10,11,14],[10,12,13,11],
....:           [12,19,15,13],[20,16,14,15],[16,20,17,2]])
sage: L.alexander_polynomial()
1
```

Some additional examples:

```
sage: B = BraidGroup(2)
sage: L = Link(B([1]))
sage: L.alexander_polynomial()
1
sage: L = Link(B.one())
sage: L.alexander_polynomial()
1
sage: B = BraidGroup(3)
sage: L = Link(B([1, 2, 1, 2]))
sage: L.alexander_polynomial()
t^-1 - 1 + t
```

When the Seifert surface is disconnected, the Alexander polynomial is defined to be 0:

```
sage: B = BraidGroup(4)
sage: L = Link(B([1,3]))
sage: L.alexander_polynomial()
0
```

TESTS:

```
sage: B = BraidGroup(4)
sage: L = Link(B([-1, 3, 1, 3]))
```

```
sage: L.alexander_polynomial()
0
sage: L = Link(B([1,3,1,1,3,3]))
sage: L.alexander_polynomial()
0
sage: B = BraidGroup(8)
sage: L = Link(B([-2, 4, 1, 6, 1, 4]))
sage: L.alexander_polynomial()
0
```

braid ( )

Return a braid representation of `self`.

OUTPUT: an element in the braid group

EXAMPLES:

```
sage: L = Link([[2, 3, 1, 4], [4, 1, 3, 2]])
sage: L.braid()
s^2
sage: L = Link([[[-1, 2, -3, 1, -2, 3]], [-1, -1, -1]])
sage: L.braid()
s^-3
sage: L = Link([[1,8,2,7], [8,4,9,5], [3,9,4,10], [10,1,7,6], [5,3,6,2]])
sage: L.braid()
(s0*s1^-1)^2*s1^-1
```

TESTS:

```
sage: L = Link([])
sage: L.braid()
1
sage: L = Link([[], []])
sage: L.braid()
1
```

determinant ( )

Return the determinant of `self`.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: L = Link(B([-1, 2, 1, 2]))
sage: L.determinant()
1
sage: B = BraidGroup(8)
sage: L = Link(B([2, 4, 2, 3, 1, 2]))
sage: L.determinant()
3
sage: L = Link(B([1]*16 + [2,1,2,1,2,2,2,2,2,2,2,1,2,1,2,-1,2,-2]))
sage: L.determinant()
65
```

TESTS:

```
sage: Link(B([1, 2, 1, -2, -1])).determinant()
Traceback (most recent call last):
...
NotImplementedError: determinant implemented only for knots
```

dowker_notation ( )

Return the Dowker notation of `self` .

Similar to the PD code we number the components, so every crossing is represented by four numbers. We focus on the incoming entities of the under and the overcrossing. It is the pair of incoming undercrossing and the incoming overcrossing. This information at every crossing gives the Dowker notation.

OUTPUT:

A list containing the pair of incoming under cross and the incoming over cross.

EXAMPLES:

```
sage: L = Link([[[-1, 2, -3, 4, 5, 1, -2, 6, 7, 3, -4, -7, -6,-5]], [-1, -1, -
↪1, -1, 1, -1, 1]])
sage: L.dowker_notation()
[(1, 6), (7, 2), (3, 10), (11, 4), (14, 5), (13, 8), (12, 9)]
sage: B = BraidGroup(4)
sage: L = Link(B([1, 2, 1, 2]))
sage: L.dowker_notation()
[(2, 1), (3, 5), (6, 4), (7, 9)]
sage: L = Link([[1, 4, 2, 3], [4, 1, 3, 2]])
sage: L.dowker_notation()
[(1, 3), (4, 2)]
```

gauss_code ( )

Return the Gauss code of `self` .

The Gauss code is generated by the following procedure:

1.Number the crossings from 1 to n.

2.Select a point on the knot and start moving along the component.

3.At each crossing, take the number of the crossing, along with sign, which is $-$ if it is an undercrossing and $+$ if it is a overcrossing.

EXAMPLES:

```
sage: L = Link([[1, 4, 2, 3], [4, 1, 3, 2]])
sage: L.gauss_code()
[[-1, 2], [1, -2]]
sage: B = BraidGroup(8)
sage: L = Link(B([1, -2, 1, -2, -2]))
sage: L.gauss_code()
[[-1, 3, -4, 5], [1, -2, 4, -5, 2, -3]]
sage: L = Link([[[-1, 2], [-3, 4], [1, 3, -4, -2]], [-1, -1, 1, 1]])
sage: L.gauss_code()
[[-1, 2], [-3, 4], [1, 3, -4, -2]]
```

genus ( )

Return the genus of `self` .

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: L = Link(B([-1, 3, 1, 3]))
sage: L.genus()
0
sage: L = Link(B([1,3]))
sage: L.genus()
0
```

```
sage: B = BraidGroup(8)
sage: L = Link(B([-2, 4, 1, 6, 1, 4]))
sage: L.genus()
0
sage: L = Link(B([1, 2, 1, 2]))
sage: L.genus()
1
```

homfly_polynomial (*var1='L'*, *var2='M'*, *normalization='lm'*)

Return the HOMFLY polynomial of `self` .

The HOMFLY polynomial $P(K)$ of a link K is a Laurent polynomial in two variables defined using skein relations and for the unknot U, we have $P(U) = 1$.

INPUT:

- `var1` – (default: `'L'`) the first variable
- `var2` – (default: `'M'`) the second variable
- `normalization` – (default: `lm`) the system of coordinates and can be one of the following:
  - `'lm'` – corresponding to the Skein relation $L \cdot P(K_+) + L^{-1} \cdot P(K_-) + M \cdot P(K_0) = 0$
  - `'az'` – corresponding to the Skein relation $a \cdot P(K_+) - a^{-1} \cdot P(K_-) = z \cdot P(K_0)$

  where $P(K_+)$, $P(K_-)$ and $P(K_0)$ represent the HOMFLY polynomials of three links that vary only in one crossing; that is the positive, negative, or smoothed links respectively

OUTPUT:

A Laurent polynomial over the integers.

Note: Use the `'az'` normalization to agree with the data in *[KnotAtlas]* and http://www.indiana.edu/~knotinfo/.

EXAMPLES:

We give some examples:

```
sage: g = BraidGroup(2).gen(0)
sage: K = Knot(g^5)
sage: K.homfly_polynomial()    # optional - libhomfly
L^-4*M^4 - 4*L^-4*M^2 + 3*L^-4 - L^-6*M^2 + 2*L^-6
```

The Hopf link:

```
sage: L = Link([[1,3,2,4],[4,2,3,1]])
sage: L.homfly_polynomial('x', 'y')  # optional - libhomfly
-x^-1*y + x^-1*y^-1 + x^-3*y^-1
```

Another version of the Hopf link where the orientation has been changed. Therefore we substitute $x \mapsto L^{-1}$ and $y \mapsto M$:

```
sage: L = Link([[1,4,2,3], [4,1,3,2]])
sage: L.homfly_polynomial()  # optional - libhomfly
L^3*M^-1 - L*M + L*M^-1
sage: L = Link([[1,4,2,3], [4,1,3,2]])
sage: L.homfly_polynomial('a', 'z', 'az')  # optional - libhomfly
a^3*z^-1 - a*z - a*z^-1
```

The figure-eight knot:

```
sage: L = Link([[2,1,4,5], [5,6,7,3], [6,4,1,9], [9,2,3,7]])
sage: L.homfly_polynomial()  # optional - libhomfly
-L^2 + M^2 - 1 - L^-2
sage: L.homfly_polynomial('a', 'z', 'az')  # optional - libhomfly
a^2 - z^2 - 1 + a^-2
```

The “monster” unknot:

```
sage: L = Link([[3,1,2,4], [8,9,1,7], [5,6,7,3], [4,18,6,5],
....:           [17,19,8,18], [9,10,11,14], [10,12,13,11],
....:           [12,19,15,13], [20,16,14,15], [16,20,17,2]])
sage: L.homfly_polynomial()  # optional - libhomfly
1
```

The knot 9_6:

```
sage: B = BraidGroup(3)
sage: K = Knot(B([-1,-1,-1,-1,-1,-1,-2,1,-2,-2]))
sage: K.homfly_polynomial()  # optional - libhomfly
L^10*M^4 - L^8*M^6 - 3*L^10*M^2 + 4*L^8*M^4 + L^6*M^6 + L^10
 - 3*L^8*M^2 - 5*L^6*M^4 - L^8 + 7*L^6*M^2 - 3*L^6
sage: K.homfly_polynomial('a', 'z', normalization='az')  # optional -␣
↪libhomfly
-a^10*z^4 + a^8*z^6 - 3*a^10*z^2 + 4*a^8*z^4 + a^6*z^6 - a^10
 + 3*a^8*z^2 + 5*a^6*z^4 - a^8 + 7*a^6*z^2 + 3*a^6
```

TESTS:

This works with isolated components:

```
sage: L = Link([[[1, -1], [2, -2]], [1, 1]])
sage: L2 = Link([[1, 3, 2, 4], [2, 3, 1, 4]])
sage: L2.homfly_polynomial()  # optional - libhomfly
-L*M^-1 - L^-1*M^-1
sage: L.homfly_polynomial()  # optional - libhomfly
-L*M^-1 - L^-1*M^-1
sage: L.homfly_polynomial('a', 'z', 'az')  # optional - libhomfly
a*z^-1 - a^-1*z^-1
sage: L2.homfly_polynomial('a', 'z', 'az')  # optional - libhomfly
a*z^-1 - a^-1*z^-1
```

REFERENCES:

- Wikipedia article HOMFLY_polynomial
- http://mathworld.wolfram.com/HOMFLYPolynomial.html

is_alternating ( )

Return `True` if the given knot diagram is alternating else returns `False` .

Alternating diagram implies every overcross is followed by an undercross or the vice-versa.

We look at the Gauss code if the sign is alternating, `True` is returned else the knot is not alternating `False` is returned.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: L = Link(B([-1, -1, -1, -1]))
```

```
sage: L.is_alternating()
False
sage: L = Link(B([1, -2, -1, 2]))
sage: L.is_alternating()
False
sage: L = Link(B([-1, 3, 1, 3, 2]))
sage: L.is_alternating()
False
sage: L = Link(B([1]*16 + [2,1,2,1,2,2,2,2,2,2,2,1,2,1,2,-1,2,-2]))
sage: L.is_alternating()
False
sage: L = Link(B([-1,2,-1,2]))
sage: L.is_alternating()
True
```

is_knot ( )

Return `True` if `self` is a knot.

Every knot is a link but the converse is not true.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: L = Link(B([1, 3, 1, -3]))
sage: L.is_knot()
False
sage: B = BraidGroup(8)
sage: L = Link(B([1, 2, 3, 4, 5, 6]))
sage: L.is_knot()
True
```

jones_polynomial (*variab=None*, *skein_normalization=False*, *algorithm='jonesrep'*)

Return the Jones polynomial of `self` .

The normalization is so that the unknot has Jones polynomial 1. If `skein_normalization` is `True` , the variable of the result is replaced by a itself to the power of 4, so that the result agrees with the conventions of [Lic] (which in particular differs slightly from the conventions used otherwise in this class), had one used the conventional Kauffman bracket variable notation directly.

If `variab` is `None` return a polynomial in the variable A or t, depending on the value `skein_normalization` . In particular, if `skein_normalization` is `False` , return the result in terms of the variable t, also used in [Lic].

ALGORITHM:

The calculation goes through one of two possible algorithms, depending on the value of `algorithm` . Possible values are `'jonesrep'` which uses the Jones representation of a braid representation of `self` to compute the polynomial of the trace closure of the braid, and `statesum` which recursively computes the Kauffman bracket of `self` . Depending on how the link is given, there might be significant time gains in using one over the other. When the trace closure of the braid is `self` , the algorithms give the same result.

INPUT:

- `variab` – variable (default: `None`); the variable in the resulting polynomial; if unspecified, use either a default variable in $\mathbf{Z}[A, A^{-1}]$ or the variable t in the symbolic ring
- `skein_normalization` – boolean (default: `False`); determines the variable of the resulting polynomial

- `algorithm` – string (default: `'jonesrep'`); algorithm to use and can be one of the following:
  - `'jonesrep'` - use the Jones representation of the braid representation
  - `'statesum'` - recursively computes the Kauffman bracket

OUTPUT:

If `skein_normalization` if `False`, this returns an element in the symbolic ring as the Jones polynomial of the link might have fractional powers when the link is not a knot. Otherwise the result is a Laurant polynomial in `variab`.

EXAMPLES:

The unknot:

```
sage: B = BraidGroup(9)
sage: b = B([1, 2, 3, 4, 5, 6, 7, 8])
sage: Link(b).jones_polynomial()
1
```

The “monster” unknot:

```
sage: L = Link([[3,1,2,4],[8,9,1,7],[5,6,7,3],[4,18,6,5],
....:           [17,19,8,18],[9,10,11,14],[10,12,13,11],
....:           [12,19,15,13],[20,16,14,15],[16,20,17,2]])
sage: L.jones_polynomial()
1
```

The Ochiai unknot:

```
sage: L = Link([[[1,-2,-3,-8,-12,13,-14,15,-7,-1,2,-4,10,11,-13,12,
....:             -11,-16,4,3,-5,6,-9,7,-15,14,16,-10,8,9,-6,5]],
....:           [-1,-1,1,1,1,1,-1,1,1,-1,1,-1,-1,-1,-1,-1]])
sage: L.jones_polynomial()   # long time
1
```

Two unlinked unknots:

```
sage: B = BraidGroup(4)
sage: b = B([1, 3])
sage: Link(b).jones_polynomial()
-sqrt(t) - 1/sqrt(t)
```

The Hopf link:

```
sage: B = BraidGroup(2)
sage: b = B([-1,-1])
sage: Link(b).jones_polynomial()
-1/sqrt(t) - 1/t^(5/2)
```

Different representations of the trefoil and one of its mirror:

```
sage: B = BraidGroup(2)
sage: b = B([-1, -1, -1])
sage: Link(b).jones_polynomial(skein_normalization=True)
-A^-16 + A^-12 + A^-4
sage: Link(b).jones_polynomial()
1/t + 1/t^3 - 1/t^4
sage: B = BraidGroup(3)
```

```
sage: b = B([-1, -2, -1, -2])
sage: Link(b).jones_polynomial(skein_normalization=True)
-A^-16 + A^-12 + A^-4
sage: R.<x> = LaurentPolynomialRing(GF(2))
sage: Link(b).jones_polynomial(skein_normalization=True, variab=x)
x^-16 + x^-12 + x^-4
sage: B = BraidGroup(3)
sage: b = B([1, 2, 1, 2])
sage: Link(b).jones_polynomial(skein_normalization=True)
A^4 + A^12 - A^16
```

$K11n42$ (the mirror of the "Kinoshita-Terasaka" knot) and $K11n34$ (the mirror of the "Conway" knot) in *[KnotAtlas]*:

```
sage: B = BraidGroup(4)
sage: K11n42 = Link(B([1, -2, 3, -2, 3, -2, -2, -1, 2, -3, -3, 2, 2]))
sage: K11n34 = Link(B([1, 1, 2, -3, 2, -3, 1, -2, -2, -3, -3]))
sage: cmp(K11n42.jones_polynomial(), K11n34.jones_polynomial())
0
```

The two algorithms for computation give the same result when the trace closure of the braid representation is the link itself:

```
sage: L = Link([[[-1, 2, -3, 4, 5, 1, -2, 6, 7, 3, -4, -7, -6, -5]],
....:           [-1, -1, -1, -1, 1, -1, 1]])
sage: jonesrep = L.jones_polynomial(algorithm='jonesrep')
sage: statesum = L.jones_polynomial(algorithm='statesum')
sage: cmp(jonesrep, statesum)
0
```

When we have thrown away unknots so that the trace closure of the braid is not necessarily the link itself, this is only true up to a power of the Jones polynomial of the unknot:

```
sage: B = BraidGroup(3)
sage: b = B([1])
sage: L = Link(b)
sage: b.components_in_closure()
2
sage: L.number_of_components()
1
sage: b.jones_polynomial()
-sqrt(t) - 1/sqrt(t)
sage: L.jones_polynomial(algorithm='statesum')
1
```

TESTS:

```
sage: L = Link([])
sage: L.jones_polynomial(algorithm='statesum')
1

sage: L.jones_polynomial(algorithm='other')
Traceback (most recent call last):
...
ValueError: bad value of algorithm
```

khovanov_homology (*ring=Integer Ring*, *height=None*, *degree=None*)

Return the Khovanov homology of the link.

INPUT:

- `ring` – (default: `ZZ`) the coefficient ring
- `height` – the height of the homology to compute, if not specified, all the heights are computed
- `degree` – the degree of the homology to compute, if not specified, all the degrees are computed

OUTPUT:

The Khovanov homology of the Link. It is given as a dictionary whose keys are the different heights. For each height, the homology is given as another dictionary whose keys are the degrees.

EXAMPLES:

```
sage: K = Link([[[1, -2, 3, -1, 2, -3]],[-1, -1, -1]])
sage: K.khovanov_homology()
{-9: {-3: Z},
 -7: {-3: 0, -2: C2},
 -5: {-3: 0, -2: Z, -1: 0, 0: 0},
 -3: {-3: 0, -2: 0, -1: 0, 0: Z},
 -1: {0: Z}}
```

The figure eight knot:

```
sage: L = Link([[1, 6, 2, 7], [5, 2, 6, 3], [3, 1, 4, 8], [7, 5, 8, 4]])
sage: L.khovanov_homology(height=-1)
{-1: {-2: 0, -1: Z, 0: Z, 1: 0, 2: 0}}
```

The Hopf link:

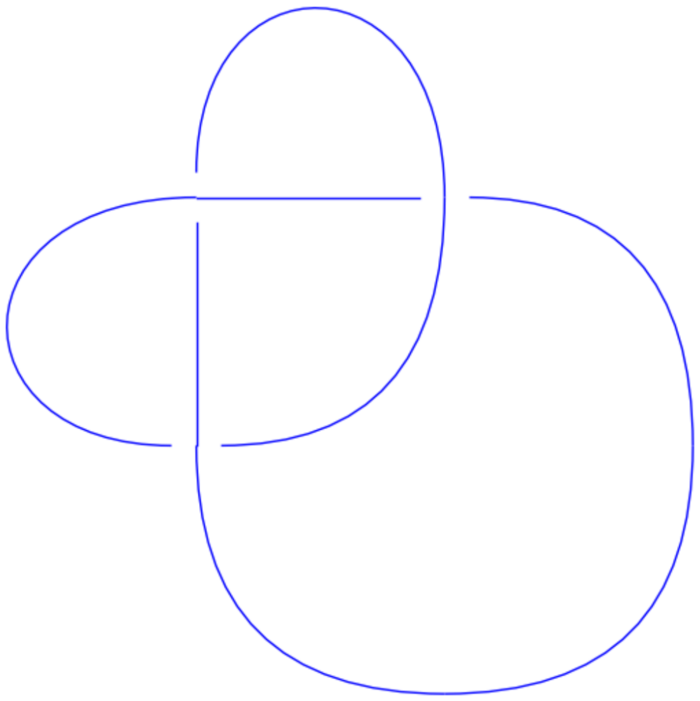
```
sage: B = BraidGroup(2)
sage: b = B([1, 1])
sage: K = Link(b)
sage: K.khovanov_homology(degree = 2)
{2: {2: 0}, 4: {2: Z}, 6: {2: Z}}
```

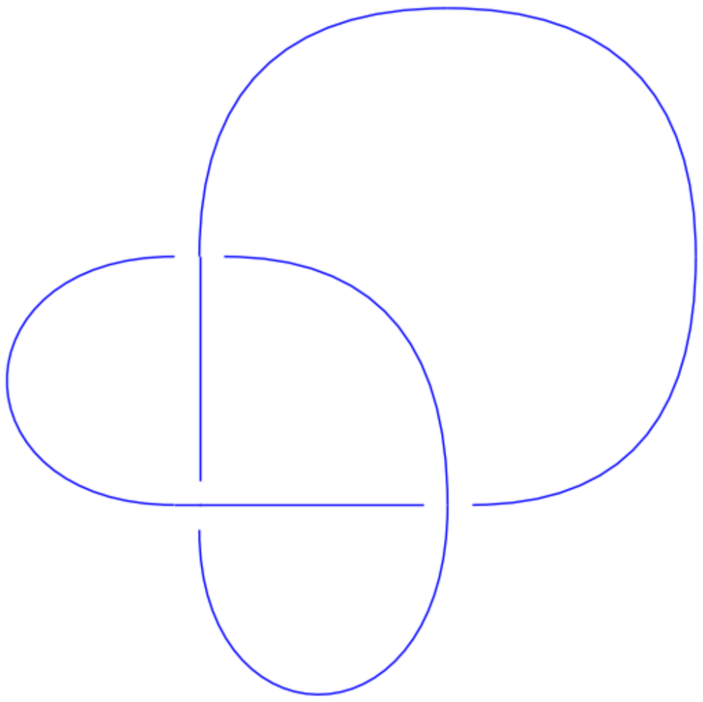
mirror_image ( )

Return the mirror image of `self` .

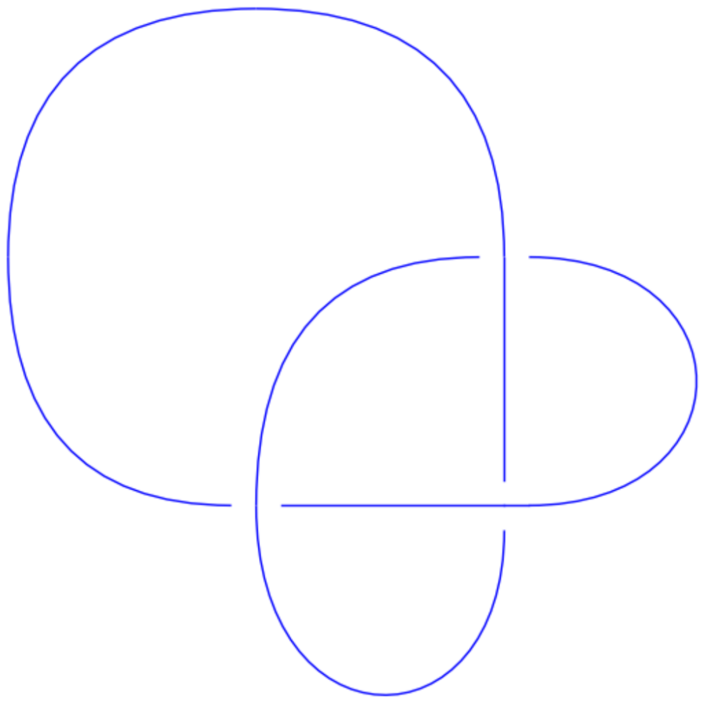
EXAMPLES:

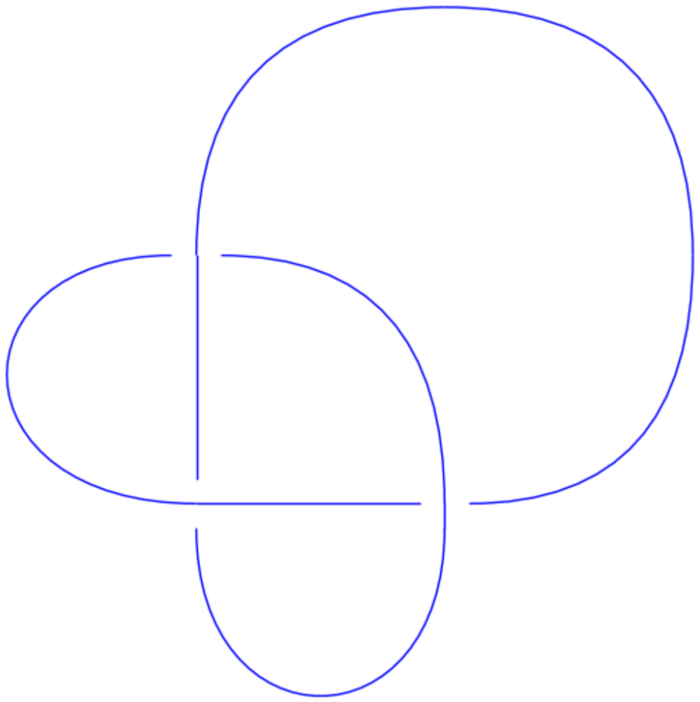
```
sage: g = BraidGroup(2).gen(0)
sage: K = Link(g^3)
sage: K2 = K.mirror_image(); K2
Link with 1 component represented by 3 crossings
sage: K2.braid()
s^-3
```





```
sage: K = Knot([[[1, -2, 3, -1, 2, -3]], [1, 1, 1]])
sage: K2 = K.mirror_image(); K2
Knot represented by 3 crossings
sage: K.pd_code()
[[4, 1, 5, 2], [2, 5, 3, 6], [6, 3, 1, 4]]
sage: K2.pd_code()
[[4, 2, 5, 1], [2, 6, 3, 5], [6, 4, 1, 3]]
```





number_of_components ( )

Return the number of connected components of `self`.

OUTPUT: number of connected components

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: L = Link(B([-1, 3, 1, 3]))
sage: L.number_of_components()
4
sage: B = BraidGroup(8)
sage: L = Link(B([-2, 4, 1, 6, 1, 4]))
sage: L.number_of_components()
5
sage: L = Link(B([1, 2, 1, 2]))
sage: L.number_of_components()
1
sage: L = Link(B.one())
sage: L.number_of_components()
1
```

orientation ( )

Return the orientation of the crossings of the link diagram of `self`.

EXAMPLES:

```
sage: L = Link([[1, 4, 5, 2], [3, 5, 6, 7], [4, 8, 9, 6], [7, 9, 10, 11], [8,␣
↪1, 13, 10], [11, 13, 2, 3]])
sage: L.orientation()
```

```
[-1, 1, -1, 1, -1, 1]
sage: L = Link([[1, 7, 2, 6], [7, 3, 8, 2], [3, 11, 4, 10], [11, 5, 12, 4],␣
↪[14, 5, 1, 6], [13, 9, 14, 8], [12, 9, 13, 10]])
sage: L.orientation()
[-1, -1, -1, -1, 1, -1, 1]
sage: L = Link([[1, 2, 3, 3], [2, 4, 5, 5], [4, 1, 7, 7]])
sage: L.orientation()
[-1, -1, -1]
```

oriented_gauss_code ( )

Return the oriented Gauss code of `self` .

The oriented Gauss code has two parts:

1.the Gauss code

2.the orientation of each crossing

The following orientation was taken into consideration for construction of knots:

From the outgoing of the overcrossing if we move in the clockwise direction to reach the outgoing of the undercrossing then we label that crossing as -1.

From the outgoing of the overcrossing if we move in the anticlockwise direction to reach the outgoing of the undercrossing then we label that crossing as $+1$.

One more consideration we take in while constructing the orientation is the order of the orientation is same as the ordering of the crossings in the Gauss code.

Note: Convention: under is denoted by -1, and over by $+1$ in the crossing info.

EXAMPLES:

```
sage: L = Link([[1, 11, 2, 10], [6, 2, 7, 3], [3, 12, 4, 9], [9, 5, 10, 6],␣
↪[8, 1, 5, 4], [11, 8, 12, 7]])
sage: L.oriented_gauss_code()
[[[-1, 2, -3, 5], [4, -2, 6, -5], [-4, 1, -6, 3]], [-1, 1, 1, 1, -1, -1]]
sage: L = Link([[1, 4, 2, 3], [6, 1, 3, 2], [7, 4, 8, 5], [5, 8, 6, 7]])
sage: L.oriented_gauss_code()
[[[-1, 2], [-3, 4], [1, 3, -4, -2]], [-1, -1, 1, 1]]
sage: B = BraidGroup(8)
sage: b = B([1, 1, 1, 1, 1])
sage: L = Link(b)
sage: L.oriented_gauss_code()
[[[1, -2, 3, -4, 5, -1, 2, -3, 4, -5]], [1, 1, 1, 1, 1]]
```

TESTS:

```
sage: L = Link([])
sage: L.oriented_gauss_code()
[[], []]
sage: L = Link(BraidGroup(2).one())
sage: L.oriented_gauss_code()
[[], []]
```

pd_code ( )

Return the planar diagram code of `self` .

The planar diagram is returned in the following format.

We construct the crossing by starting with the entering component of the undercrossing, move in the clockwise direction and then generate the list. If the crossing is given by $[a, b, c, d]$, then we interpret this information as:

1. a is the entering component of the undercrossing;
2. b, d are the components of the overcrossing;
3. c is the leaving component of the undercrossing.

EXAMPLES:

```
sage: L = Link([[[1, -2, 3, -4, 2, -1, 4, -3]], [1, 1, -1, -1]])
sage: L.pd_code()
[[6, 1, 7, 2], [2, 5, 3, 6], [8, 4, 1, 3], [4, 8, 5, 7]]
sage: B = BraidGroup(2)
sage: b = B([1, 1, 1, 1, 1])
sage: L = Link(b)
sage: L.pd_code()
[[2, 1, 3, 4], [4, 3, 5, 6], [6, 5, 7, 8], [8, 7, 9, 10], [10, 9, 1, 2]]
sage: L = Link([[[2, -1], [1, -2]], [1, 1]])
sage: L.pd_code()
[[2, 3, 1, 4], [4, 1, 3, 2]]
sage: L = Link([[1, 2, 3, 3], [2, 4, 5, 5], [4, 1, 7, 7]])
sage: L.pd_code()
[[1, 2, 3, 3], [2, 4, 5, 5], [4, 1, 7, 7]]
```

TESTS:

```
sage: L = Link([[], []])
sage: L.pd_code()
[]
sage: L = Link(BraidGroup(2).one())
sage: L.pd_code()
[]
```

plot (*gap=0.1*, *component_gap=0.5*, *solver=None*, ***kwargs*)

Plot `self`.

INPUT:

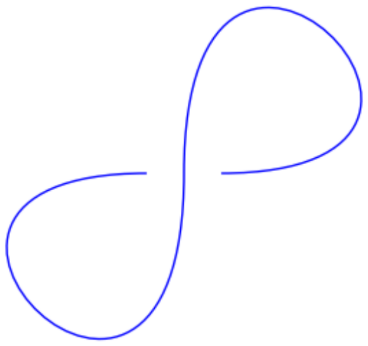
- `gap` – (default: 0.1) the size of the blank gap left for the crossings
- `component_gap` – (default: 0.5) the gap between isolated components
- `solver` – the linear solver to use, see `MixedIntegerLinearProgram`.

The usual keywords for plots can be used here too.

EXAMPLES:

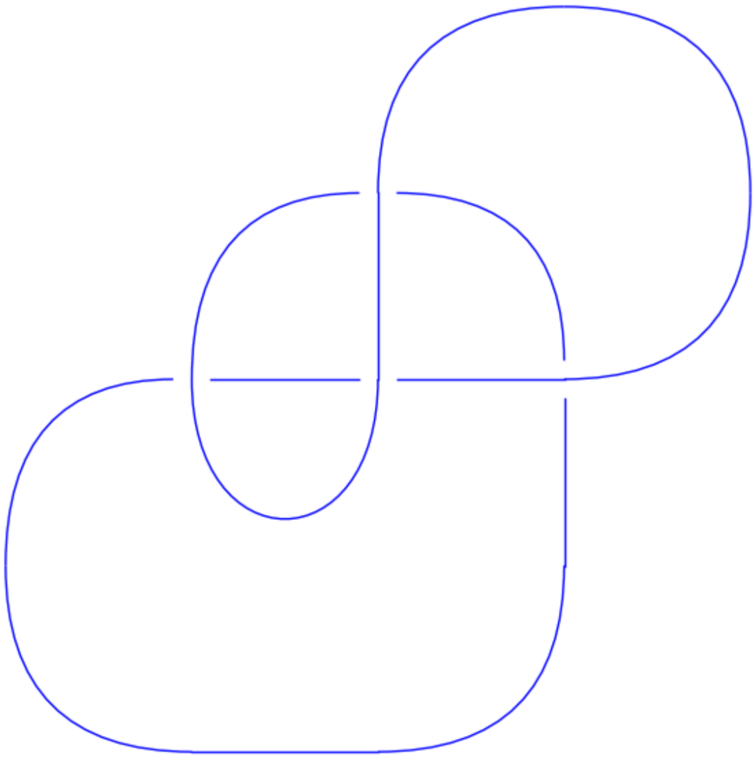
We construct the simplest version of the unknot:

```
sage: L = Link([[2, 1, 1, 2]])
sage: L.plot()
Graphics object consisting of ... graphics primitives
```



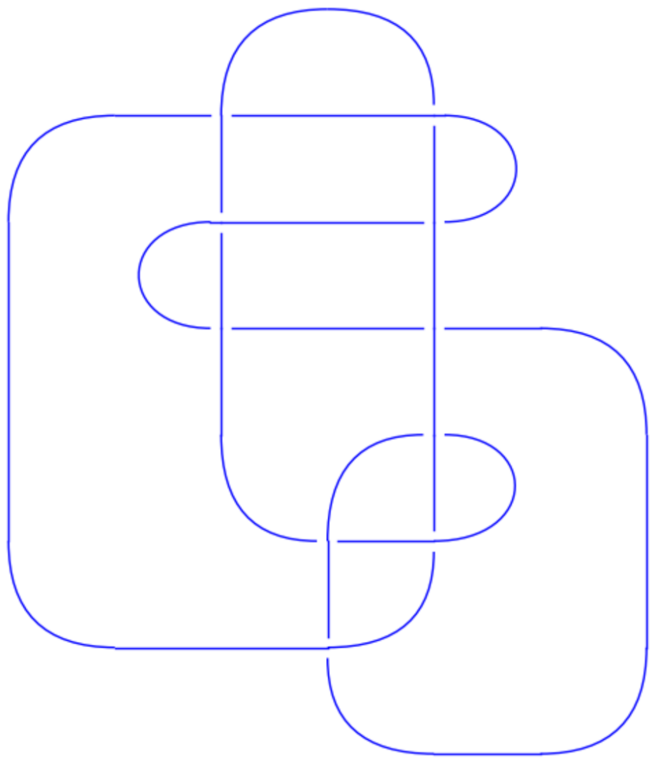
We construct a more interesting example of the unknot:

```
sage: L = Link([[2, 1, 4, 5], [3, 5, 6, 7], [4, 1, 9, 6], [9, 2, 3, 7]])
sage: L.plot()
Graphics object consisting of ... graphics primitives
```



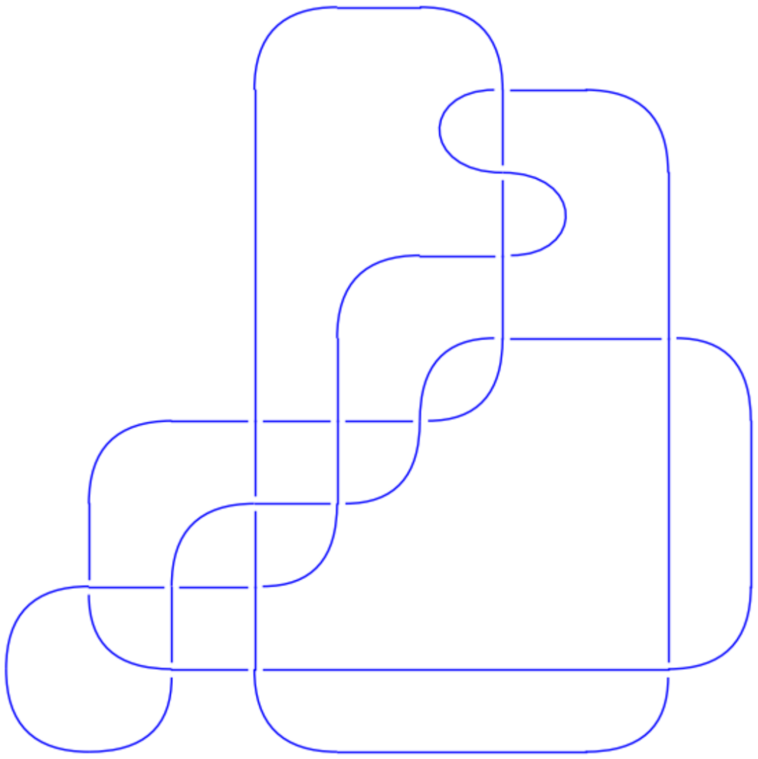
The "monster" unknot:

```
sage: L = Link([[3,1,2,4],[8,9,1,7],[5,6,7,3],[4,18,6,5],
....:           [17,19,8,18],[9,10,11,14],[10,12,13,11],
....:           [12,19,15,13],[20,16,14,15],[16,20,17,2]])
sage: L.plot()
Graphics object consisting of ... graphics primitives
```



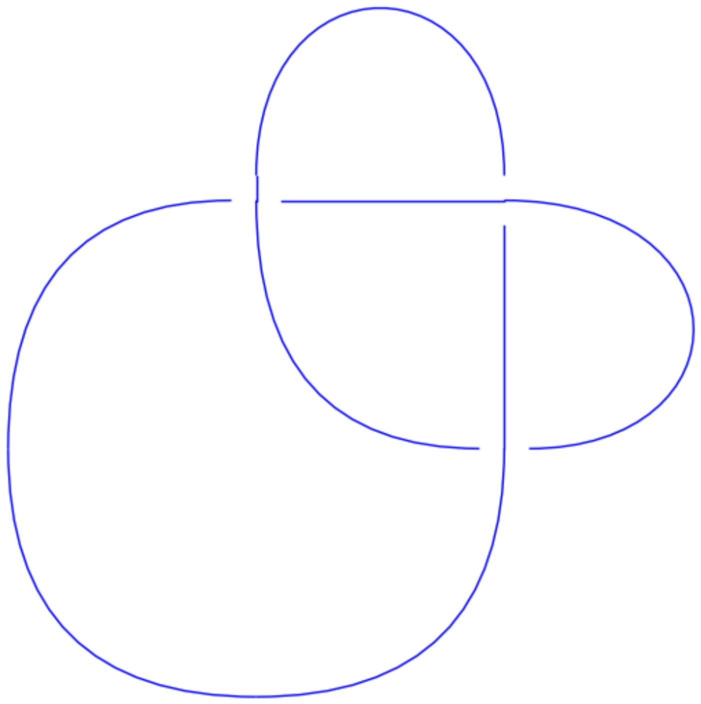
The Ochiai unknot:

```
sage: L = Link([[[1,-2,-3,-8,-12,13,-14,15,-7,-1,2,-4,10,11,-13,12,
....:             -11,-16,4,3,-5,6,-9,7,-15,14,16,-10,8,9,-6,5]],
....:           [-1,-1,1,1,1,1,-1,1,1,-1,1,-1,-1,-1,-1,-1]])
sage: L.plot()
Graphics object consisting of ... graphics primitives
```



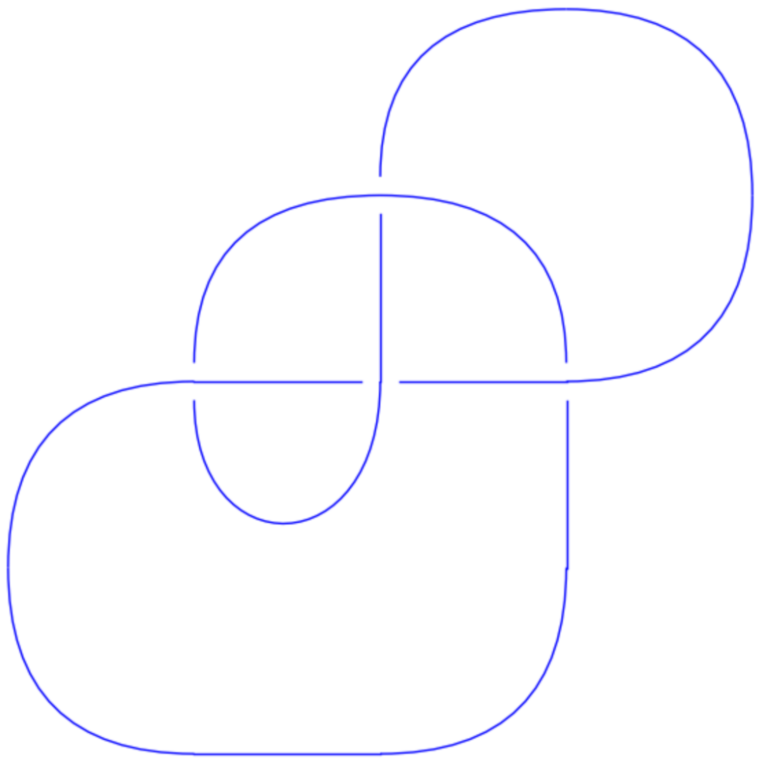
One of the representations of the trefoil knot:

```
sage: L = Link([[1, 5, 2, 4], [5, 3, 6, 2], [3, 1, 4, 6]])
sage: L.plot()
Graphics object consisting of 14 graphics primitives
```



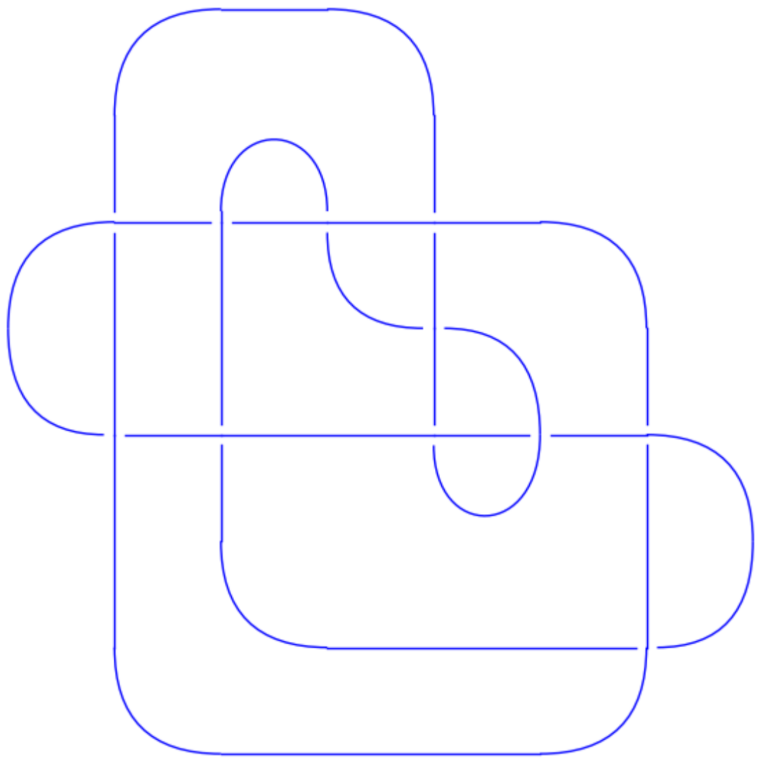
The figure-eight knot:

```
sage: L = Link([[2, 1, 4, 5], [5, 6, 7, 3], [6, 4, 1, 9], [9, 2, 3, 7]])
sage: L.plot()
Graphics object consisting of ... graphics primitives
```



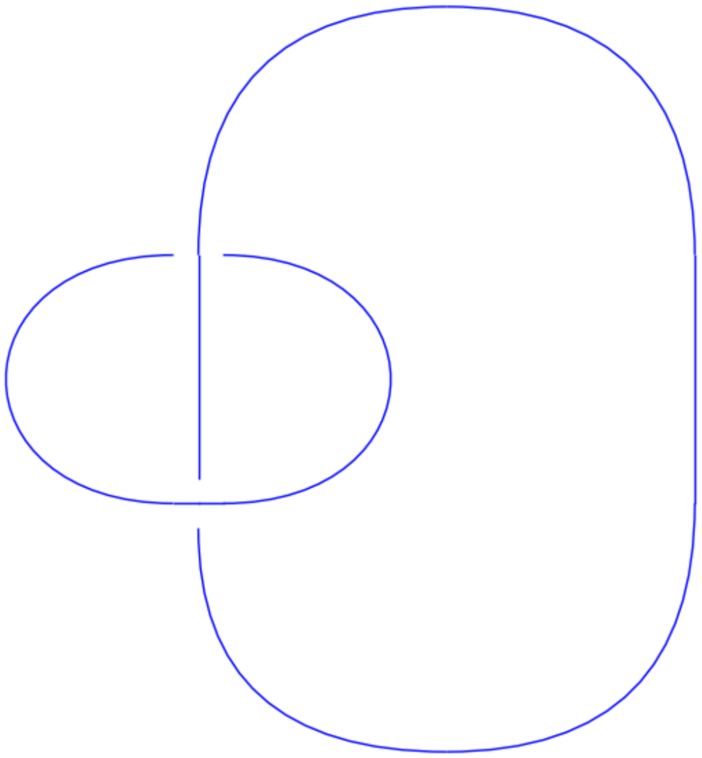
The knot $K11n121$ in *[KnotAtlas]*:

```
sage: L = Link([[4,2,5,1], [10,3,11,4], [5,16,6,17], [7,12,8,13],
....:           [18,9,19,10], [2,11,3,12], [13,20,14,21], [15,6,16,7],
....:           [22,18,1,17], [8,19,9,20], [21,14,22,15]])
sage: L.plot()
Graphics object consisting of ... graphics primitives
```



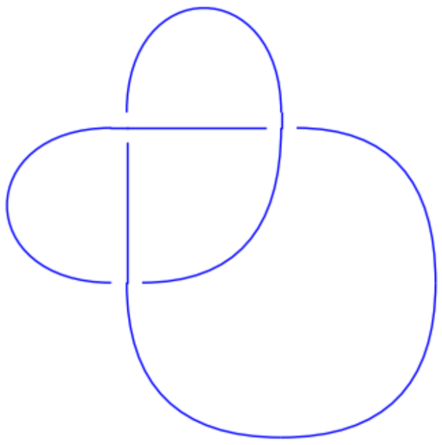
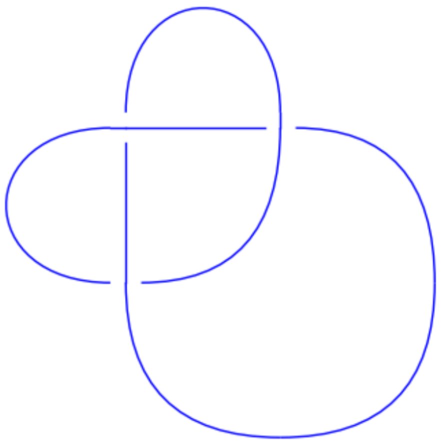
One of the representations of the Hopf link:

```
sage: L = Link([[1, 4, 2, 3], [4, 1, 3, 2]])
sage: L.plot()
Graphics object consisting of ... graphics primitives
```



Plotting links with multiple isolated components:

```
sage: L = Link([[[-1, 2, -3, 1, -2, 3], [4, -5, 6, -4, 5, -6]], [1, 1, 1, 1,␣
↪1, 1]])
sage: L.plot()
Graphics object consisting of ... graphics primitives
```

TESTS:

Check that trac ticket #20315 is fixed:

```
sage: L = Link([[2,1,4,5], [5,6,7,3], [6,4,1,9], [9,2,3,7]])
sage: L.plot(solver='GLPK')
Graphics object consisting of ... graphics primitives
sage: L.plot(solver='Coin')     # optional - cbc
Graphics object consisting of ... graphics primitives
sage: L.plot(solver='CPLEX')    # optional - CPLEX
Graphics object consisting of ... graphics primitives
sage: L.plot(solver='Gurobi')   # optional - Gurobi
Graphics object consisting of ... graphics primitives
```

regions ( )

Return the regions from the link diagram of `self` .

Regions are obtained always turning left at each crossing.

Then the regions are represented as a list with the segments that form its boundary, with a sign depending on the orientation of the segment as part of the boundary.

EXAMPLES:

```
sage: L = Link([[[-1, +2, -3, 4, +5, +1, -2, +6, +7, 3, -4, -7, -6,-5]],[-1, -
↪1, -1, -1, 1, -1, 1]])
sage: L.regions()
[[1, 7, 3, 11, 5], [2, -7], [4, -11], [6, -1], [8, -13, 10, -3], [9, 13],␣
↪[12, -9, 14, -5], [-14, -8, -2, -6], [-12, -4, -10]]
sage: L = Link([[[1, -2, 3, -4, 2, -1, 4, -3]],[1, 1, -1, -1]])
```

```
sage: L.regions()
[[1, 7, -4], [2, -5, -7], [3, -8, 5], [4, 8], [6, -1, -3], [-2, -6]]
sage: L = Link([[[-1, +2, 3, -4, 5, -6, 7, 8, -2, -5, +6, +1, -8, -3, 4, -
↪7]],[-1, -1, -1, -1, 1, 1, -1, 1]])
sage: L.regions()
[[1, 13, -8], [2, -9, -13], [3, -14, 9], [4, 16, 8, 14], [5, 11, 7, -16], [6,␣
↪-11], [10, -5, -15, -3], [12, -1, -7], [15, -4], [-12, -6, -10, -2]]
sage: B = BraidGroup(2)
sage: L = Link(B([-1, -1, -1]))
sage: L.regions()
[[1, 3, 5], [2, -1], [4, -3], [6, -5], [-2, -6, -4]]
sage: L = Link([[[1, -2, 3, -4], [-1, 5, -3, 2, -5, 4]], [-1, 1, 1, -1, -1]])
sage: L.regions()
[[1, -5], [2, -8, 4, 5], [3, 8], [6, -9, -2], [7, -3, 9], [10, -4, -7], [-10,␣
↪-6, -1]]
sage: L = Link([[1, 2, 3, 3], [2, 5, 4, 4], [5, 7, 6, 6], [7, 1, 8, 8]])
sage: L.regions()
[[-3], [-4], [-6], [-8], [1, 2, 5, 7], [-2, 3, -1, 8, -7, 6, -5, 4]]
```

Note: The link diagram is assumed to have only one completely isolated component. This is because otherwise some regions would have disconnected boundary.

TESTS:

```
sage: B = BraidGroup(6)
sage: L = Link(B([1, 3, 5]))
sage: L.regions()
Traceback (most recent call last):
...
NotImplementedError: can only have one isolated component
```

seifert_circles ()

Return the Seifert circles from the link diagram of `self`.

Seifert circles are the circles obtained by smoothing all crossings respecting the orientation of the segments.

Each Seifert circle is represented as a list of the segments that form it.

EXAMPLES:

```
sage: L = Link([[[1, -2, 3, -4, 2, -1, 4, -3]], [1, 1, -1, -1]])
sage: L.seifert_circles()
[[1, 7, 5, 3], [2, 6], [4, 8]]
sage: L = Link([[[-1, 2, 3, -4, 5, -6, 7, 8, -2, -5, 6, 1, -8, -3, 4, -7]], [-
↪1, -1, -1, -1, 1, 1, -1, 1]])
sage: L.seifert_circles()
[[1, 13, 9, 3, 15, 5, 11, 7], [2, 10, 6, 12], [4, 16, 8, 14]]
sage: L = Link([[[-1, 2, -3, 4, 5, 1, -2, 6, 7, 3, -4, -7, -6,-5]], [-1, -1, -
↪1, -1, 1, -1, 1]])
sage: L.seifert_circles()
[[1, 7, 3, 11, 5], [2, 8, 14, 6], [4, 12, 10], [9, 13]]
sage: L = Link([[1, 7, 2, 6], [7, 3, 8, 2], [3, 11, 4, 10], [11, 5, 12, 4],␣
↪[14, 5, 1, 6], [13, 9, 14, 8], [12, 9, 13, 10]])
sage: L.seifert_circles()
[[1, 7, 3, 11, 5], [2, 8, 14, 6], [4, 12, 10], [9, 13]]
sage: L = Link([[[-1, 2, -3, 5], [4, -2, 6, -5], [-4, 1, -6, 3]], [-1, 1, 1,␣
↪1, -1, -1]])
```

```
sage: L.seifert_circles()
[[1, 11, 8], [2, 7, 12, 4, 5, 10], [3, 9, 6]]
sage: B = BraidGroup(2)
sage: L = Link(B([1, 1, 1]))
sage: L.seifert_circles()
[[1, 3, 5], [2, 4, 6]]
```

seifert_matrix ( )

Return the Seifert matrix associated with `self` .

ALGORITHM:

This is the algorithm presented in Section 3.3 of *[Collins13]*.

OUTPUT:

The intersection matrix of a (not necessarily minimal) Seifert surface.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: L = Link(B([-1, 3, 1, 3]))
sage: L.seifert_matrix()
[ 0  0]
[ 0 -1]
sage: B = BraidGroup(8)
sage: L = Link(B([-1, 3, 1, 5, 1, 7, 1, 6]))
sage: L.seifert_matrix()
[ 0  0  0]
[ 1 -1  0]
[ 0  1 -1]
sage: L = Link(B([-2, 4, 1, 6, 1, 4]))
sage: L.seifert_matrix()
[-1  0]
[ 0 -1]
```

signature ( )

Return the signature of `self` .

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: L = Link(B([-1, 3, 1, 3]))
sage: L.signature()
-1
sage: B = BraidGroup(8)
sage: L = Link(B([-2, 4, 1, 6, 1, 4]))
sage: L.signature()
-2
sage: L = Link(B([1, 2, 1, 2]))
sage: L.signature()
-2
```

writhe ( )

Return the writhe of `self` .

EXAMPLES:

```
sage: L = Link([[[1, -2, 3, -4, 2, -1, 4, -3]],[1, 1, -1, -1]])
sage: L.writhe()
```

```
0
sage: L = Link([[[-1, 2, -3, 4, 5, 1, -2, 6, 7, 3, -4, -7, -6,-5]],
....:             [-1, -1, -1, -1, 1, -1, 1]])
sage: L.writhe()
-3
sage: L = Link([[[-1, 2, 3, -4, 5, -6, 7, 8, -2, -5, 6, 1, -8, -3, 4, -7]],
....:             [-1, -1, -1, -1, 1, 1, -1, 1]])
sage: L.writhe()
-2
```

CHAPTER

THREE

INDICES AND TABLES

- Index
- Module Index
- Search Page

[Collins13] Julia Collins. *An algorithm for computing the Seifert matrix of a link from a braid representation*. (2013). http://www.maths.ed.ac.uk/~jcollins/SeifertMatrix/SeifertMatrix.pdf

[KnotAtlas] The Knot atlas. http://katlas.org/wiki/Main_Page

PYTHON MODULE INDEX

INDEX