<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cellular Automata</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.1 Soliton Cellular Automata</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Plotting of Mandelbrot and Julia Sets</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>2.1 Mandelbrot and Julia sets</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Abelian differentials and flat surfaces</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>3.1 Strata of differentials on Riemann surfaces</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.2 Strata of quadratic differentials on Riemann surfaces</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Interval exchange transformations and linear involutions</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>4.1 Class factories for Interval exchange transformations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.2 Labelled permutations</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>4.3 Reduced permutations</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>4.4 Permutations template</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>4.5 Interval Exchange Transformations and Linear Involution</td>
<td>109</td>
</tr>
<tr>
<td>5</td>
<td>Sandpiles</td>
<td>117</td>
</tr>
<tr>
<td>6</td>
<td>Arithmetic Dynamical Systems</td>
<td>171</td>
</tr>
<tr>
<td></td>
<td>6.1 Generic dynamical systems on schemes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.2 Dynamical systems on affine schemes</td>
<td>177</td>
</tr>
<tr>
<td></td>
<td>6.3 Dynamical systems on projective schemes</td>
<td>187</td>
</tr>
<tr>
<td></td>
<td>6.4 Dynamical systems for products of projective spaces</td>
<td>239</td>
</tr>
<tr>
<td></td>
<td>6.5 Wehler K3 Surfaces</td>
<td>241</td>
</tr>
<tr>
<td>7</td>
<td>Indices and Tables</td>
<td>263</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>265</td>
</tr>
<tr>
<td></td>
<td>Python Module Index</td>
<td>267</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>269</td>
</tr>
</tbody>
</table>
1.1 Soliton Cellular Automata

AUTHORS:

- Travis Scrimshaw (2017-06-30): Initial version
- Travis Scrimshaw (2018-02-03): Periodic version

```python
class sage.dynamics.cellular_automata.solitons.PeriodicSolitonCellularAutomata(initial_state, cartan_type=2, vacuum=1)
```

Bases: `sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata`

A periodic soliton cellular automata.

Fix some $r \in I_0$. A periodic soliton cellular automata is a `SolitonCellularAutomata` with a state being a fixed number of tensor factors $p = p_\ell \otimes \cdots \otimes p_1 \otimes p_0$ and the time evolution $T_s$ is defined by

$$R(p \otimes u) = u \otimes T_s(p),$$

for some element $u \in B^{r,s}$.

INPUT:

- `initial_state` – the list of elements, can also be a string when vacuum is 1 and $n$ is $s\ell_n$
- `cartan_type` – (default: 2) the value $n$, for $s\ell_n$, or a Cartan type
- `r` – (default: 1) the node index $r$; typically this corresponds to the height of the vacuum element

EXAMPLES:

The construction and usage is the same as for `SolitonCellularAutomata`:

```python
sage: P = PeriodicSolitonCellularAutomata('112334111241111423111411123112', 4)
sage: P.evolve()
sage: P
```

Soliton cellular automata of type ['$A', 3, 1] and vacuum = 1

initial state:

..23334...24....423...4...23..2
evoluions: [(1, 31)]
current state:

34......24....243....4.223.233.

```python
sage: P.evolve(carrier_capacity=2)
```

(continues on next page)
Using $r = 2$ in type $A_3^{(1)}$:

```plaintext
sage: initial = [[2,1], [2,1], [2,1], [2,1], [2,1], [2,1], [3,1], [3,1], [3,2]]
sage: P = PeriodicSolitonCellularAutomata(initial, 4, 2)
sage: P.print_states(10)
t: 0 4 333 ..1...112
t: 1 4 333 .1.112...
t: 2 433 3 112.....1
t: 3 3 334 2....111.
t: 4 334 3 ..111...2
t: 5 34 33 11.....21
t: 6 333 11....1121.
t: 7 333 4 .112..1..
t: 8 3 4 33 2....1 11
t: 9 3433 .1112...
```

We do some examples in other types:

```plaintext
sage: initial = [[1],[2],[2],[1],[1],[1],[3],[1],[3,1],[3,1],[3,2]]
sage: P = PeriodicSolitonCellularAutomata(initial, 4, 2)
sage: P.print_states(10)
t: 0 4 333 ..1...112
t: 1 4 333 .1.112...
t: 2 433 3 112.....1
t: 3 3 334 2....111.
t: 4 334 3 ..111...2
t: 5 34 33 11.....21
t: 6 333 11....1121.
t: 7 333 4 .112..1..
t: 8 3 4 33 2....1 11
t: 9 3433 .1112...
```

(continues on next page)
### References

- [KTT2006]
- [KS2006]
- [YT2002]
- [YYT2003]

#### evolve (carrier_capacity=None, carrier_index=None, number=None)

Evolve self.

Time evolution $T_s$ of a SCA state $p$ is determined by

$$ u \otimes T_s(p) = R(p \otimes u), $$

where $u$ is some element in $B^{r,*}$.

**Input:**

- `carrier_capacity` – (default: the number of balls in the system) the size $s$ of carrier
- `carrier_index` – (default: the vacuum index) the index $r$ of the carrier
- `number` – (optional) the number of times to perform the evolutions

To perform multiple evolutions of the SCA, `carrier_capacity` and `carrier_index` may be lists of the same length.

**Warning:** Time evolution is only guaranteed to result in a solution when the `carrier_index` is the defining $r$ of the SCA. If no solution is found, then this will raise an error.
EXAMPLES:

```python
sage: P = PeriodicSolitonCellularAutomata('12411321431221122', 4)
sage: P.evolve()
.sage: P.print_state(0)
.24...332.4.3.22..22
sage: P.print_state(1)
4...33.2.42322..22..
sage: P.evolve(carrier_capacity=2)
.sage: P.print_state(2)
..33.22.4232..22...4
sage: P.evolve(carrier_capacity=[1,3,1,2])
sage: P.evolve(1, number=3)
sage: P.print_states(10)
t: 0
 .24...332.4.3.22..22
t: 1
 4...33.2.42322..22..
t: 2
 ..33.22.4232..22...4
t: 3
 .33.22.4232..22...4.
t: 4
 3222..43.2.22....4.3
t: 5
 222..43.2.22....4.33
t: 6
 2...4322.2......43322
t: 7
 ...4322.2.....433222
t: 8
 ..4322.2.....433222.
t: 9
 .4322.2.....433222.
sage: P = PeriodicSolitonCellularAutomata('1241132121', 4)
sage: P.evolve(carrier_index=2, carrier_capacity=3)
sage: P.state_evolution(0)
[[[1, 1, 1], [2, 2, 4]],
 [[1, 1, 2], [2, 2, 4]],
 [[1, 1, 3], [2, 2, 4]],
 [[1, 1, 1], [2, 2, 3]],
 [[1, 1, 1], [2, 2, 3]],
 [[1, 1, 1], [2, 2, 3]],
 [[1, 1, 2], [2, 2, 3]],
 [[1, 1, 1], [2, 2, 2]],
 [[1, 1, 1], [2, 2, 2]],
 [[1, 1, 1], [2, 2, 2]],
 [[1, 1, 1], [2, 2, 4]],
 [[1, 1, 1], [2, 2, 4]]]
```

class sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata (initial_state, carrier_type=2, vacuum=1)

Bases: sage.structure.sage_object.SageObject
Soliton cellular automata.
Fix an affine Lie algebra $g$ with index $I$ and classical index set $I_0$. Fix some $r \in I_0$. A soliton cellular automaton (SCA) is a discrete (non-linear) dynamical system given as follows. The states are given by elements of a semi-infinite tensor product of Kirillov-Reshetikhin crystals $B^{r,1}$, where only a finite number of factors are not the maximal element $u$, which we will call the vacuum. The time evolution $T_s$ is defined by

$$R(p \otimes u_s) = u_s \otimes T_s(p),$$

where $p = \cdots \otimes p_3 \otimes p_2 \otimes p_1 \otimes p_0$ is a state and $u_s$ is the maximal element of $B^{r,s}$. In more detail, we have $R(p_i \otimes u(i)) = u(i+1) \otimes \bar{p}_i$ with $u(0) = u_s$ and $T_s(p) = \cdots \otimes \bar{p}_1 \otimes \bar{p}_0$. This is well-defined since $R(u \otimes u_s) = u_s \otimes u$ and $u(k) = u_s$ for all $k \gg 1$.

**INPUT:**
- initial_state – the list of elements, can also be a string when vacuum is 1 and $n$ is $sl_n$
- cartan_type – (default: 2) the value $n$, for $sl_n$, or a Cartan type
- $r$ – (default: 1) the node index $r$; typically this corresponds to the height of the vacuum element

**EXAMPLES:**
We first create an example in $sl_4$ (type $A_3$):

```
sage: B = SolitonCellularAutomata('34111112241112223', 4)
sage: B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
initial state:
34......224....2223
evolutions: []
current state:
34......224....2223
```

We then apply an standard evolution:

```
sage: B.evolve()
sage: B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
initial state:
34......224....2223
evolutions: [(1, 19)]
current state:
...............34....224...2223....
```

Next, we apply a smaller carrier evolution. Note that the soliton of size 4 moves only 3 steps:

```
sage: B.evolve(3)
sage: B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
initial state:
34......224....2223
evolutions: [(1, 19), (1, 3)]
current state:
.................34......224...2223....
```

We can also use carriers corresponding to non-vacuum indices. In these cases, the carrier might not return to its initial state, which results in a message being displayed about the resulting state of the carrier:

```
sage: B.evolve(carrier_capacity=7, carrier_index=3)
Last carrier:
(continues on next page)
```
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1

Initial state:
34......224.....2223

Evolution: [(1, 19), (1, 3), (3, 7)]

Current state:
......................23....222....2223......

sage: B.evolve(carrier_capacity=3, carrier_index=2)

Last carrier:
1 1 1
2 2 3

Soliton cellular automata of type ['A', 3, 1] and vacuum = 1

Initial state:
34......224.....2223

Evolution: [(1, 19), (1, 3), (3, 7), (2, 3)]

Current state:
.......................22.....223...2222........

To summarize our current evolutions, we can use `print_states()`:

```
sage: B.print_states(5)
t: 0
......................34......224.....2223

t: 1
......................34......224...2223....

t: 2
......................34....224...2223.......

t: 3
......................23....222....2223.......

t: 4
......................22.....223...2222........
```

To run the SCA further under the standard evolutions, one can use `print_states()` or `latex_states()`:

```
sage: B.print_states(15)
t: 0
........................34......224....2223

t: 1
........................34....224...2223....

t: 2
........................34....224...2223.......

t: 3
........................23....222....2223.......

t: 4
........................22.....223...2222........

t: 5
........................22....223..2222............

t: 6
........................22..2223..222................

t: 7
........................2222..23...222................

t: 8
........................22222..23...2222...........
```

(continues on next page)
Next, we use $r = 2$ in type $A_3$. Here, we give the data as lists of values corresponding to the entries of the column of height 2 from the largest entry to smallest. Our columns are drawn in French convention:

$$\text{sage: } B = \text{SolitonCellularAutomata}\left([\begin{array}{c} 4, 1 \\ 4, 1 \\ 2, 1 \\ 2, 1 \\ 2, 1 \\ 2, 1 \\ 3, 1 \\ 3, 1 \\ 3, 2 \end{array}], 4, 2\right)$$

We perform 3 evolutions and obtain the following:

$$\text{sage: } B.\text{evolve}(\text{number}=3)$$
$$\text{sage: } B$$
Soliton cellular automata of type ['A', 3, 1] and vacuum = 2
initial state:
44 333
11...112
evolutions: [(2, 9), (2, 9), (2, 9)]
current state:
44 333
...11.112.......

We construct Example 2.9 from [LS2017]:

$$\text{sage: } B = \text{SolitonCellularAutomata}\left([\begin{array}{c} 2 \\ -3 \\ 1 \\ 1 \\ 1 \\ 4 \\ 0 \\ -2 \\ \vdots \\ 1, 1, 1, 1, 1, 1, 3, -4, -3, -3, 1 \end{array}], ['D', 5, 2]\right)$$

We perform 3 evolutions and obtain the following:

$$\text{sage: } B.\text{print}\_\text{states}(10)$$
$$\text{sage: } B$$
1.1. Soliton Cellular Automata
Example 3.4 from [LS2017]:

\begin{verbatim}
 sage: B = SolitonCellularAutomata([['E'],[1],[1],[1],[3],[0],
 ....: [1],[1],[1],[1],[2],[-3],[-1],[1]], ['D',4,2])
 sage: B.print_states(10)
\end{verbatim}

Example 3.12 from [LS2017]:

\begin{verbatim}
 sage: B = SolitonCellularAutomata([[-1,3,2],[3,2,1],[3,2,1],[-3,2,1],
 ....: [-2,-3,1]], ['B',3,1], 3)
 sage: B.print_states(6)
\end{verbatim}

Example 4.12 from [LS2017]:

\begin{verbatim}
 sage: K = crystals.KirillovReshetikhin(['E',6,1], 1,1, 'KR')
 sage: u = K.module_generators[0]
\end{verbatim}
sage: x = u.f_string([1,3,4,5])
sage: y = u.f_string([1,3,4,2,5,6])
sage: a = u.f_string([1,3,4,2])
sage: B = SolitonCellularAutomata([a, u,u,u, x,y], ['E',6,1], 1)
sage: B
Soliton cellular automata of type ['E', 6, 1] and vacuum = 1
  initial state:
  (-2, 5) . . . (-5, 2, 6)(-2, -6, 4)
  evoluitions: []
  current state:
  (-2, 5) . . . (-5, 2, 6)(-2, -6, 4)
sage: B.print_states(8)
t: 0 
  . (-2, 5)(-2, -5, 4, 6) ... (-6, 2) ...
t: 7

**evolve** (carrier_capacity=None, carrier_index=None, number=None)

Evolve self.

Time evolution $T_s$ of a SCA state $p$ is determined by

$$u_{r,s} \otimes T_s(p) = R(p \otimes u_{r,s}),$$

where $u_{r,s}$ is the maximal element of $B^{r,s}$.

**INPUT:**

- carrier_capacity – (default: the number of balls in the system) the size $s$ of carrier
- carrier_index – (default: the vacuum index) the index $r$ of the carrier
- number – (optional) the number of times to perform the evolutions

To perform multiple evolutions of the SCA, carrier_capacity and carrier_index may be lists of the same length.

**EXAMPLES:**

sage: B = SolitonCellularAutomata('3411111122411112223', 4)
sage: for k in range(10):
    ....: B.evolve()
  ....: B.evolve()
sage: B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
  initial state:
  34......224....2223
  evoluitions: [(1, 19), (1, 19), (1, 19), (1, 19), (1, 19),
              (1, 19), (1, 19), (1, 19), (1, 19), (1, 19)]
  current state:
  ......2344.......222....23.............................

sage: B.reset()
sage: B.evolve(number=10); B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
  initial state:
  34......224....2223
  evoluitions: [(1, 19), (1, 19), (1, 19), (1, 19), (1, 19),
              (1, 19), (1, 19), (1, 19), (1, 19), (1, 19)]
  current state:
  ......2344.......222....23.............................
sage: B.reset()
sage: B.evolve(carrier_capacity=[1,2,3,4,5,6,7,8,9,10]); B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
initial state:
34......224....223
evolutuions: [(1, 1), (1, 2), (1, 3), (1, 4), (1, 5),
(1, 6), (1, 7), (1, 8), (1, 9), (1, 10)]
current state:
........2344....222..23..............................
sage: B.reset()
sage: B.evolve(carrier_index=[1,2,3])
Last carrier:
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 4 4
sage: B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
initial state:
34......224....223
evolutuions: [(1, 19), (2, 19), (3, 19)]
current state:
..........................22......223...2222....
sage: B.reset()
sage: B.evolve(carrier_capacity=[1,2,3], carrier_index=[1,2,3])
Last carrier:
1 1
3 4
Last carrier:
1 1 1
2 2 3
3 3 4
sage: B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
initial state:
34......224....223
evolutuions: [(1, 1), (2, 2), (3, 3)]
current state:
.....22.......223....2222..
sage: B.reset()
sage: B.evolve(1, 2, number=3)
Last carrier:
1
3
Last carrier:
1
4
Last carrier:
1
3
sage: B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
initial state:
34......224....223
evolutuions: [(2, 1), (2, 1), (2, 1)]

(continues on next page)
latex_state_evolution \( (\text{num}, \text{scale}=1) \)
Return a latex version of the evolution process of the state \( \text{num} \).

See also:

state_evolution(), print_state_evolution()

EXAMPLES:

```
sage: B = SolitonCellularAutomata('113123', 3)
sage: B.evolve(3)
sage: B.latex_state_evolution(0)
\begin{tikzpicture}[scale=1]
  \node (i0) at (0.0,0.9) {$1$};
  \node (i1) at (2.48,0.9) {$1$};
  \node (i2) at (4.96,0.9) {$3$};
  \...
  \draw[->] (i5) -- (t5);
  \draw[->] (u6) -- (u5);
\end{tikzpicture}
sage: B.latex_state_evolution(1)
\begin{tikzpicture}[scale=1]
  ...
\end{tikzpicture}
```

latex_states \( (\text{num}=\text{None}, \text{as_array}=\text{True}, \text{box_width}=\'5pt\') \)
Return a latex version of the states.

INPUT:

• \( \text{num} \) – the number of states

• \( \text{as_array} \) (default: True) if True, then the states are placed inside of an array; if False, then the states are given as a word

• \( \text{box_width} \) – (default: '5pt') the width of the . used to represent the vacuum state when \( \text{as_array} \) is True

If \( \text{as_array} \) is False, then the vacuum element is printed in a gray color. If \( \text{as_array} \) is True, then the vacuum is given as .

Use the \( \text{box_width} \) to help create more even spacing when a column in the output contains only vacuum elements.

EXAMPLES:

```
sage: B = SolitonCellularAutomata('411122', 4)
sage: B.latex_states(8)
\begin{array}{c|ccccccccccccccccccc}
  t = 0 & \cdots & ... & \makebox[5pt]{.} & 4 & \makebox[5pt]{.} & \makebox[5pt]{.} & 2 & 2 \\
  t = 1 & \cdots & ... & 4 & \makebox[5pt]{.} & \makebox[5pt]{.} & 2 & 2 & ... \\
  t = 2 & \cdots & ... & 4 & \makebox[5pt]{.} & 2 & 2 & 2 & 2 \& ... \\
  t = 3 & \cdots & ... & 4 & 2 & 4 & 2 & 2 & 2 & ... \\
  t = 4 & \cdots & ... & 2 & 4 & 2 & 2 & 2 & 2 & ... \\
  t = 5 & \cdots & ... & 2 & 4 & \makebox[5pt]{.} & 2 & 2 & ... \\
\end{array}
```

(continues on next page)
print_state (num=None, vacuum_letter='.', remove_trailing_vacuums=False)
Print the state num.

INPUT:

• num – (default: the current state) the state to print
• vacuum_letter – (default: '.') the letter to print for the vacuum
• remove_trailing_vacuums – (default: False) if True then this does not print the vacuum letters at the right end of the state

EXAMPLES:

sage: B = SolitonCellularAutomata('3411111122411112223', 4)
sage: B.print_state()
34......224....2223

print_state_evolution (num)
Print the evolution process of the state num.

See also:

state_evolution(), latex_state_evolution()

EXAMPLES:

sage: B = SolitonCellularAutomata('1113123', 3)
sage: B.evolve(3)
sage: B.print_state_evolution(0)
print_states \( \texttt{(num=None, vacuum_letter='.'\}) \)

Print the first \texttt{num} states of \texttt{self}.

**Note:** If the number of states computed for \texttt{self} is less than \texttt{num}, then this evolves the system using the default time evolution.

**INPUT:**

- \texttt{num} – the number of states to print

**EXAMPLES:**

```python
sage: B = SolitonCellularAutomata([[2],[-1],[1],[1],[1],[1],[2],[2],[3],
.....: [2],[-1],[1],[2],[-1],[1],[1],[1],[1],[1],[2],[3],[3],[-3],[-2]],
.....: ['C',3,1])
sage: B.print_states(7)
t: 0 ..........................21....2232..21......23332
    ________________
t: 1 ..........................21...2232...21....23332....
    ________________
t: 2 ..........................21..2232....21..23332..........
    ________________
t: 3 ..........................221..232...2231..332............
    ________________
t: 4 ..........................221...232.2231....332..................
    ________________
t: 5 ..........................221...2321223......332..................
    ________________
t: 6 ..........................221...2321223332....................
    ________________

sage: B = SolitonCellularAutomata([[2],[1],[1],[1],[3],[-2],[1],[1],
.....: [1],[2],[2],[-3],[1],[1],[1],[1],[1],[1],[2],[3],[3],[-3]],
.....: ['B',3,1])
sage: B.print_states(9, ' ')
t: 0 ..........................2 32 223 2333
    ________________
t: 1 ..........................2 32 223 2333
    ________________
t: 2 ..........................2 32 223 2333
    ________________
t: 3 ..........................23 2223 2333
    ________________
t: 4 ..........................23 2223 2333
    ________________
```

(continues on next page)
Example 4.13 from [Yamada2007]:

```
sage: B = SolitonCellularAutomata([[2],[-2],[1],[1],[1],[1],[2],[0],[-3],[1],[1],[1],[1],[1],[2],[2],[3],[-3]],["D",4,2])
sage: B.print_states(15)
```

```
t: 0
.........................22...203....2233
  
t: 1                         22..203...2233...1
  
t: 2                      22..203...2233       
  
t: 3                   22.203..2233.......   
  
t: 4                     22203.2233........  
  
t: 5                     220223.233........  
  
t: 6                  2202.223.33................
  
t: 7                        2202..223..33...........
  
t: 8                               2202...223...33....................
  
t: 9                                           2202....223....33..........................
```

Example 4.13 from [Yamada2007]:

```
sage: B = SolitonCellularAutomata([[3],[3],[1],[1],[1],[1],[2],[2],[2]],["D",4,3])
sage: B.print_states(15)
```

```
t: 0
.........................33...222
  
t: 1                         33...222...
  
t: 2                      33...222.....   
  
t: 3                   33.222........      
  
t: 4                     33.222.........    
  
t: 5                     33222............   
  
t: 6                  3022...............        
  
t: 7                        332...........
  
t: 8                               3E..................        
```
Example 4.14 from [Yamada2007]:

```
sage: B = SolitonCellularAutomata([[3], [1], [1], [1], [2], [3], [1], [1], [1], [2], [3], [3]], ['D', 4, 3])
sage: B.print_states(15)
t: 0 ..................................3...23...233
t: 1 ..................................3..23..233...
t: 2 ..................................3.23.233....
t: 3 ..................................323233....
t: 4 ..................................0033.........
t: 5 .................................._313.........
t: 6 ..................................30E.3.........
t: 7 .................................._333...3.....
t: 8 ..................................3302...3.....
t: 9 ..................................33322.....3....
t: 10 ..................................333222.....3..
t: 11 ..................................333..22......3..
t: 12 ..................................333...22......3..
t: 13 ..................................333....22.....3..
t: 14 ..................................333.....22....3...
reset()
Reset self back to the initial state.
```

1.1. Soliton Cellular Automata
### state_evolution(\texttt{num})

Return a list of the carrier values at state \texttt{num} evolving to the next state.

If \texttt{num} is greater than the number of states, this performs the standard evolution $T_k$, where $k$ is the number of balls in the system.

See also:

\texttt{print_state_evolution()}, \texttt{latex_state_evolution()}

**EXAMPLES:**

```python
sage: B = SolitonCellularAutomata('1113123', 3)
sage: B.evolve(3)
sage: B.state_evolution(0)
[[[1, 1, 1]],
 [[1, 1, 1]],
 [[1, 1, 1]],
 [[1, 1, 3]],
 [[1, 1, 2]],
 [[1, 2, 3]],
 [[1, 1, 3]],
 [[1, 1, 1]]]
sage: B.state_evolution(2)
[[[1, 1, 1, 1, 1, 1, 1]],
 [[1, 1, 1, 1, 1, 1, 1]],
 [[1, 1, 1, 1, 1, 1, 1]],
 [[1, 1, 1, 1, 1, 1, 1]],
 [[1, 1, 1, 1, 1, 1, 1]],
 [[1, 1, 1, 1, 3, 3]],
 [[1, 1, 1, 1, 1, 3]],
 [[1, 1, 1, 1, 3]]
```

(continues on next page)
[[1, 1, 1, 1, 1, 1, 2]],
[[1, 1, 1, 1, 1, 1, 1]],
[[1, 1, 1, 1, 1, 1, 1]],
[[1, 1, 1, 1, 1, 1, 1]],
[[1, 1, 1, 1, 1, 1, 1]],
[[1, 1, 1, 1, 1, 1, 1]],
[[1, 1, 1, 1, 1, 1, 1]],
[[1, 1, 1, 1, 1, 1, 1]]

1.1. Soliton Cellular Automata
CHAPTER TWO

PLOTTING OF MANDELBROT AND JULIA SETS

2.1 Mandelbrot and Julia sets

Plots the Mandelbrot and Julia sets for the map \( Q_c(z) = z^2 + c \) in the complex plane.

The Mandelbrot set is the set of complex numbers \( c \) for which the function \( Q_c(z) = z^2 + c \) does not diverge when iterated from \( z = 0 \). This set of complex numbers can be visualized by plotting each value for \( c \) in the complex plane. The Mandelbrot set is an example of a fractal when plotted in the complex plane.

The Julia set for a given \( c \) is the set of complex numbers for which the function \( Q_c(z) = z^2 + c \) is bounded under iteration.

AUTHORS:
- Ben Barros

sage.dynamics.complex_dynamics.mandel_julia.external_ray(theta, **kwds)

Draws the external ray(s) of a given angle (or list of angles) by connecting a finite number of points that were approximated using Newton’s method. The algorithm used is described in a paper by Tomoki Kawahira.

REFERENCE:
[Kaw2009]

INPUT:
- theta – double or list of doubles, angles between 0 and 1 inclusive.

kwds:
- image – 24-bit RGB image (optional - default: None) user specified image of Mandelbrot set.
- D – long (optional - default: 25) depth of the approximation. As \( D \) increases, the external ray gets closer to the boundary of the Mandelbrot set. If the ray doesn’t reach the boundary of the Mandelbrot set, increase \( D \).
- S – long (optional - default: 10) sharpness of the approximation. Adjusts the number of points used to approximate the external ray (number of points is equal to \( S \times D \)). If ray looks jagged, increase \( S \).
- R – long (optional - default: 100) radial parameter. If \( R \) is large, the external ray reaches sufficiently close to infinity. If \( R \) is too small, Newton’s method may not converge to the correct ray.
- prec – long (optional - default: 300) specifies the bits of precision used by the Complex Field when using Newton’s method to compute points on the external ray.
- ray_color – RGB color (optional - default: [255, 255, 255]) color of the external ray(s).

OUTPUT:
24-bit RGB image of external ray(s) on the Mandelbrot set.
EXAMPLES:

```python
sage: external_ray(1/3)
500x500px 24-bit RGB image

sage: external_ray(0.6, ray_color=[255, 0, 0])
500x500px 24-bit RGB image

sage: external_ray([0, 0.2, 0.4, 0.7])
500x500px 24-bit RGB image

sage: external_ray([i/5 for i in range(1,5)])
500x500px 24-bit RGB image
```

WARNING:

If you are passing in an image, make sure you specify which parameters to use when drawing the external ray. For example, the following is incorrect:

```python
sage: M = mandelbrot_plot(x_center=0)  # not tested
sage: external_ray(5/7, image=M)      # not tested
500x500px 24-bit RGB image
```

To get the correct external ray, we adjust our parameters:

```python
sage: M = mandelbrot_plot(x_center=0)
sage: external_ray(5/7, x_center=0, image=M)
500x500px 24-bit RGB image
```

Todo: The copy() function for bitmap images needs to be implemented in Sage.

```python
sage.dynamics.complex_dynamics.mandel_julia.julia_plot(c=-1,
                                                      x_center=0.0,
y_center=0.0,
image_width=4.0,
max_iteration=500,
pixel_count=500,
base_color='steelblue',
iteration_level=1,
number_of_colors=50,
point_color='yellow',
interact=False,
mandelbrot=True, period=None)
```

Plots the Julia set of a given complex $c$ value. Users can specify whether they would like to display the Mandelbrot side by side with the Julia set.

The Julia set of a given $c$ value is the set of complex numbers for which the function $Q_c(z) = z^2 + c$ is bounded under iteration. The Julia set can be visualized by plotting each point in the set in the complex plane. Julia sets are examples of fractals when plotted in the complex plane.

**ALGORITHM:**

Define the map $Q_c(z) = z^2 + c$ for some $c \in \mathbb{C}$. For every $p \in \mathbb{C}$, if $|Q^k_c(p)| > 2$ for some $k \geq 0$, then $Q^k_c(p) \to \infty$. Let $N$ be the maximum number of iterations. Compute the first $N$ points on the orbit of $p$ under $Q_c$. If for any $k < N$, $|Q^k_c(p)| > 2$, we stop the iteration and assign a color to the point $p$ based on how quickly $p$ escaped to infinity under iteration of $Q_c$. If $|Q^i_c(p)| \leq 2$ for all $i \leq N$, we assume $p$ is in the Julia set and assign the point $p$ the color black.
INPUT:

- **c** – complex (optional - default: `-1`), complex point \( c \) that determines the Julia set.
- **period** – list (optional - default: `None`), returns the Julia set for a random \( c \) value with the given (formal) cycle structure.
- **mandelbrot** – boolean (optional - default: `True`), when set to `True`, an image of the Mandelbrot set is appended to the right of the Julia set.
- **point_color** – RGB color (optional - default: `'tomato'`), color of the point \( c \) in the Mandelbrot set (any valid input for Color).
- **x_center** – double (optional - default: `-1.0`), Real part of center point.
- **y_center** – double (optional - default: `0.0`), Imaginary part of center point.
- **image_width** – double (optional - default: `4.0`), width of image in the complex plane.
- **max_iteration** – long (optional - default: `500`), maximum number of iterations the map \( Q_c(z) \).
- **pixel_count** – long (optional - default: `500`), side length of image in number of pixels.
- **base_color** – RGB color (optional - default: `'steelblue'`), color used to determine the coloring of set (any valid input for Color).
- **iteration_level** – long (optional - default: `1`), number of iterations between each color level.
- **number_of_colors** – long (optional - default: `30`), number of colors used to plot image.
- **interact** – boolean (optional - default: `False`), controls whether plot will have interactive functionality.

OUTPUT:

24-bit RGB image of the Julia set in the complex plane.

EXAMPLES:

```plaintext
sage: julia_plot()
1001x500px 24-bit RGB image
```

To display only the Julia set, set `mandelbrot` to `False`:

```plaintext
sage: julia_plot(mandelbrot=False)
500x500px 24-bit RGB image
```

To display an interactive plot of the Julia set in the Notebook, set `interact` to `True`:

```plaintext
sage: julia_plot(interact=True)
interactive(children=(FloatSlider(value=-1.0, description=u'Real c'...
```

To return the Julia set of a random \( c \) value with (formal) cycle structure \((2, 3)\), set `period = [2, 3]`:

```plaintext
sage: julia_plot(period=[2,3])
1001x500px 24-bit RGB image
```

To return all of the Julia sets of \( c \) values with (formal) cycle structure \((2, 3)\):

```plaintext
sage: period = [2,3]  # not tested
....: R.<c> = QQ[]
....: P.<x,y> = ProjectiveSpace(R,1)
....: f = DynamicalSystem([x^2+c*y^2, y^2])
(continues on next page)
```

2.1. Mandelbrot and Julia sets
....: L = f.dynatomic_polynomial(period).subs({x:0,y:1}).roots(ring=CC)
....: c_values = [k[0] for k in L]
....: for c in c_values:
....:    julia_plot(c)

sage.dynamics.complex_dynamics.mandel_julia.mandelbrot_plot(x_center=-1.0, y_center=0.0, image_width=4.0, max_iteration=500, pixel_count=500, base_color='steelblue', iteration_level=1, number_of_colors=30, interact=False)

Interactive plot of the Mandelbrot set for the map $Q_c(z) = z^2 + c$.  

**ALGORITHM:**

Let each pixel in the image be a point $c \in \mathbb{C}$ and define the map $Q_c(z) = z^2 + c$. If $|Q^k_c(c)| > 2$ for some $k \geq 0$, it follows that $Q^n_c(c) \to \infty$. Let $N$ be the maximum number of iterations. Compute the first $N$ points on the orbit of 0 under $Q_c$. If for any $k < N$, $|Q^k_c(0)| > 2$, we stop the iteration and assign a color to the point $c$ based on how quickly 0 escaped to infinity under iteration of $Q_c$. If $|Q^i_c(0)| \leq 2$ for all $i \leq N$, we assume $c$ is in the Mandelbrot set and assign the point $c$ the color black.

**REFERENCE:**

[Dev2005]

**INPUT:**

- `x_center` – double (optional - default: -1.0), Real part of center point.
- `y_center` – double (optional - default: 0.0), Imaginary part of center point.
- `image_width` – double (optional - default: 4.0), width of image in the complex plane.
- `max_iteration` – long (optional - default: 500), maximum number of iterations the map $Q_c(z)$.
- `pixel_count` – long (optional - default: 500), side length of image in number of pixels.
- `base_color` – RGB color (optional - default: 'steelblue') color used to determine the coloring of set (any valid input for Color).
- `iteration_level` – long (optional - default: 1) number of iterations between each color level.
- `number_of_colors` – long (optional - default: 30) number of colors used to plot image.
- `interact` – boolean (optional - default: False), controls whether plot will have interactive functionality.

**OUTPUT:**

24-bit RGB image of the Mandelbrot set in the complex plane.

**EXAMPLES:**

```
sage: mandelbrot_plot()
500x500px 24-bit RGB image
```
To display an interactive plot of the Mandelbrot set in the Jupyter Notebook, set `interact` to `True`:

```python
sage: manelbrot_plot(interact=True)
```

```python
sage: manelbrot_plot(interact=True, x_center=-0.75, y_center=0.25, 
               image_width=1/2, number_of_colors=75)
```
3.1 Strata of differentials on Riemann surfaces

**Warning:** This module is deprecated. You are advised to install and use the surface_dynamics package instead available at https://pypi.python.org/pypi/surface_dynamics/

The space of Abelian (or quadratic) differentials is stratified by the degrees of the zeroes (and simple poles for quadratic differentials). Each stratum has one, two or three connected components and each is associated to an (extended) Rauzy class. The `connected_components()` method (only available for Abelian stratum) give the decomposition of a stratum (which corresponds to the SAGE object `AbelianStratum`).

The work for Abelian differentials was done by Maxim Kontsevich and Anton Zorich in [KZ2003] and for quadratic differentials by Erwan Lanneau in [Lan2008]. Zorich gave an algorithm to pass from a connected component of a stratum to the associated Rauzy class (for both interval exchange transformations and linear involutions) in [Zor2008] and is implemented for Abelian stratum at different level (approximately one for each component):

- for connected stratum `representative()`
- for hyperelliptic component `representative()`
- for non hyperelliptic component, the algorithm is the same as for connected component
- for odd component `representative()`
- for even component `representative()`

The inverse operation (pass from an interval exchange transformation to the connected component) is partially written in [KZ2003] and simply named here `connected_component()`.

All the code here was first available on Mathematica [Zor].

**Note:** The quadratic strata are not yet implemented.

**AUTHORS:**

- Vincent Delecroix (2009-09-29): initial version

**EXAMPLES:**

Construction of a stratum from a list of singularity degrees:

```sage
d = AbelianStratum(1,1)
doctest:warning
```
DeprecationWarning: AbelianStratum is deprecated and will be removed from Sage. You are advised to install the surface_dynamics package via:
sage -pip install surface_dynamics
If you do not have write access to the Sage installation you can alternatively do
sage -pip install surface_dynamics --user
The package surface_dynamics subsumes all flat surface related computation that are currently available in Sage. See more information at http://www.labri.fr/perso/vdelecro/surface-dynamics/latest/
See http://trac.sagemath.org/20695 for details.

sage: a
H(1, 1)
sage: a.genus()
2
sage: a.nintervals()
5

sage: a = AbelianStratum(4,3,2,1)
sage: a
H(4, 3, 2, 1)
sage: a.genus()
6
sage: a.nintervals()
15

By convention, the degrees are always written in decreasing order:

sage: a1 = AbelianStratum(4,3,2,1)
sage: a1
H(4, 3, 2, 1)
sage: a2 = AbelianStratum(2,3,1,4)
sage: a2
H(4, 3, 2, 1)
sage: a1 == a2
True

It is also possible to consider stratum with an incoming or an outgoing separatrix marked (the aim of this consideration is to attach a specified degree at the left or the right of the associated interval exchange transformation):

sage: a_out = AbelianStratum(1, 1, marked_separatrix='out')
sage: a_out
H^out(1, 1)
sage: a_in = AbelianStratum(1, 1, marked_separatrix='in')
sage: a_in
H^in(1, 1)
sage: a_out == a_in
False

Get a list of strata with constraints on genus or on the number of intervals of a representative:

sage: for a in AbelianStrata(genus=3):
    ....:     print(a)
doctest:warning
...
DeprecationWarning: AbelianStrata is deprecated and will be removed from Sage. You are advised to install the surface_dynamics package via:
   sage -pip install surface_dynamics
If you do not have write access to the Sage installation you can alternatively do
   sage -pip install surface_dynamics --user
The package surface_dynamics subsumes all flat surface related computation that are currently available in Sage. See more information at
   http://www.labri.fr/perso/vdelecro/surface-dynamics/latest/
See http://trac.sagemath.org/20695 for details.

H(4)
H(3, 1)
H(2, 2)
H(2, 1, 1)
H(1, 1, 1, 1)

sage: for a in AbelianStrata(nintervals=5):
    ....:   print(a)
    H^out(0, 2)
    H^out(2, 0)
    H^out(1, 1)
    H^out(0, 0, 0)

sage: for a in AbelianStrata(genus=2, nintervals=5):
    ....:   print(a)
    H^out(0, 2)
    H^out(2, 0)
    H^out(1, 1)

Obtains the connected components of a stratum:

sage: a = AbelianStratum(0)
sage: a.connected_components()
[H_hyp(0)]

sage: a = AbelianStratum(6)
sage: cc = a.connected_components()
sage: cc
[H_hyp(6), H_odd(6), H_even(6)]
sage: for c in cc:
    ....:     print(c)
    ....:     print(c.representative(alphabet=range(1,9)))
H_hyp(6)
1 2 3 4 5 6 7 8
8 7 6 5 4 3 2 1
H_odd(6)
1 2 3 4 5 6 7 8
4 3 6 5 8 7 2 1
H_even(6)
1 2 3 4 5 6 7 8
6 5 4 3 8 7 2 1

sage: a = AbelianStratum(1, 1, 1, 1)
sage: a.connected_components()
The zero attached on the left of the associated Abelian permutation corresponds to the first singularity degree:

```
sage: a = AbelianStratum(4, 2, marked_separatrix='out')
sage: b = AbelianStratum(2, 4, marked_separatrix='out')
sage: a == b
False
```

The representatives are hence different:

```
sage: a_odd, a_even = a.connected_components()
sage: b_odd, b_even = b.connected_components()
sage: a_odd.representative(alphabet=range(1,10))
1 2 3 4 5 6 7 8 9
4 3 6 5 7 9 8 2 1
sage: b_odd.representative(alphabet=range(1,10))
1 2 3 4 5 6 7 8 9
4 3 5 7 6 9 8 2 1
sage: a_even.representative(alphabet=range(1,10))
1 2 3 4 5 6 7 8 9
6 5 4 3 7 9 8 2 1
sage: b_even.representative(alphabet=range(1,10))
1 2 3 4 5 6 7 8 9
7 6 5 4 3 9 8 2 1
```

You can retrieve the decomposition of the irreducible Abelian permutations into Rauzy diagrams from the classification of strata:

```
sage: a = AbelianStrata(nintervals=4)
sage: l = sum([stratum.connected_components() for stratum in a], [])
sage: n = [x.rauzy_diagram().cardinality() for x in l]
sage: for c,i in zip(l,n):
    ....:     print("{} : {}").format(c, i)
H_hyp^out(2) : 7
H_hyp^out(0, 0, 0) : 6
sage: sum(n)
13
```

```
sage: a = AbelianStrata(nintervals=5)
sage: l = sum([stratum.connected_components() for stratum in a], [])
sage: n = [x.rauzy_diagram().cardinality() for x in l]
sage: for c,i in zip(l,n):
    ....:     print("{} : {}").format(c, i)
H_hyp^out(0, 2) : 11
H_hyp^out(2, 0) : 35
H_hyp^out(1, 1) : 15
```

Abelian strata.

INPUT:

- genus - a non negative integer or None
- nintervals - a non negative integer or None
- marked_separatrix - ‘no’ (for no marking), ‘in’ (for marking an incoming separatrix) or ‘out’ (for marking an outgoing separatrix)

EXAMPLES:

Abelian strata with a given genus:

```python
sage: for s in AbelianStrata(genus=1): print(s)
H(0)
```

```python
sage: for s in AbelianStrata(genus=2): print(s)
H(2)
H(1, 1)
```

```python
sage: for s in AbelianStrata(genus=3): print(s)
H(4)
H(3, 1)
H(2, 2)
H(2, 1, 1)
H(1, 1, 1, 1)
```

```python
sage: for s in AbelianStrata(genus=4): print(s)
H(6)
H(5, 1)
H(4, 2)
H(4, 1, 1)
H(3, 3)
H(3, 2, 1)
```

(continues on next page)
Abelian strata with a given number of intervals:

```python
sage: for s in AbelianStrata(nintervals=2): print(s)
H^out(0)
```

```python
sage: for s in AbelianStrata(nintervals=3): print(s)
H^out(0, 0)
```

```python
sage: for s in AbelianStrata(nintervals=4): print(s)
H^out(0, 0, 0)
```

```python
sage: for s in AbelianStrata(nintervals=5): print(s)
H^out(0, 2)
```

Abelian strata with both constraints:

```python
sage: for s in AbelianStrata(genus=2, nintervals=4): print(s)
H^out(2)
```

```python
sage: for s in AbelianStrata(genus=5, nintervals=12): print(s)
H^out(8, 0, 0)
```

```python
H^out(0, 8, 0)
```

```python
H^out(0, 7, 1)
```

```python
H^out(1, 7, 0)
```

```python
H^out(7, 1, 0)
```

```python
H^out(0, 6, 2)
```

```python
H^out(2, 6, 0)
```

```python
H^out(6, 2, 0)
```

```python
H^out(1, 6, 1)
```

```python
H^out(6, 1, 1)
```

```python
H^out(0, 5, 3)
```

```python
H^out(3, 5, 0)
```

```python
H^out(5, 3, 0)
```

```python
H^out(1, 5, 2)
```

```python
H^out(2, 5, 1)
```

```python
H^out(5, 2, 1)
```

```python
H^out(0, 4, 4)
```

```python
H^out(4, 4, 0)
```

```python
H^out(1, 4, 3)
```

```python
H^out(3, 4, 1)
```

```python
H^out(4, 3, 1)
```

```python
H^out(2, 4, 2)
```

```python
H^out(4, 2, 2)
```

```python
H^out(2, 3, 3)
```

```python
H^out(3, 3, 2)
```
class sage.dynamics.flat_surfaces.strata.AbelianStrata_all
    Bases: sage.combinat.combinat.InfiniteAbstractCombinatorialClass

Abelian strata.

class sage.dynamics.flat_surfaces.strata.AbelianStrata_d
    Bases: sage.combinat.combinat.CombinatorialClass

Strata with constraint number of intervals.

INPUT:
    • nintervals - an integer greater than 1
    • marked_separatrix - ‘no’, ‘out’ or ‘in’

class sage.dynamics.flat_surfaces.strata.AbelianStrata_g
    Bases: sage.combinat.combinat.CombinatorialClass

Stratas of genus g surfaces.

INPUT:
    • genus - a non negative integer
    • marked_separatrix - ‘no’, ‘out’ or ‘in’

class sage.dynamics.flat_surfaces.strata.AbelianStrata_gd
    Bases: sage.combinat.combinat.CombinatorialClass

Abelian strata of prescribed genus and number of intervals.

INPUT:
    • genus - integer: the genus of the surfaces
    • nintervals - integer: the number of intervals
    • marked_separatrix - ‘no’, ‘in’ or ‘out’

class sage.dynamics.flat_surfaces.strata.AbelianStratum(*l, **d)
    Bases: sage.structure.sage_object.SageObject

Stratum of Abelian differentials.

A stratum with a marked outgoing separatrix corresponds to Rauzy diagram with left induction, a stratum with marked incoming separatrix correspond to Rauzy diagram with right induction. If there is no marked separatrix, the associated Rauzy diagram is the extended Rauzy diagram (consideration of the sage.dynamics.interval_exchanges.template.Permutation.symmetric() operation of Boissy-Lanneau).

When you want to specify a marked separatrix, the degree on which it is the first term of your degrees list.

INPUT:
    • marked_separatrix - None (default) or ‘in’ (for incoming separatrix) or ‘out’ (for outgoing separatrix).

EXAMPLES:
Creation of an Abelian stratum and get its connected components:
sage: a = AbelianStratum(2, 2)
sage: a
H(2, 2)
sage: a.connected_components()
[H_hyp(2, 2), H_odd(2, 2)]

Specification of marked separatrix:

sage: a = AbelianStratum(4, 2, marked_separatrix='in')
sage: a
H^in(4, 2)
sage: b = AbelianStratum(2, 4, marked_separatrix='in')
sage: b
H^in(2, 4)
sage: a == b
False
sage: a = AbelianStratum(4, 2, marked_separatrix='out')
sage: a
H^out(4, 2)
sage: b = AbelianStratum(2, 4, marked_separatrix='out')
sage: b
H^out(2, 4)
sage: a == b
False

Get a representative of a connected component:

sage: a = AbelianStratum(2, 2)
sage: a_hyp, a_odd = a.connected_components()
sage: a_hyp.representative()
1 2 3 4 5 6 7
7 6 5 4 3 2 1
sage: a_odd.representative()
0 1 2 3 4 5 6
3 2 4 6 5 1 0

You can choose the alphabet:

sage: a_odd.representative(alphabet="ABCDEFGHIJKLMNOPQRSTUVWXYZ")
A B C D E F G
D C E G F B A

By default, you get a reduced permutation, but you can specify that you want a labelled one:

sage: p_reduced = a_odd.representative()
sage: p_labelled = a_odd.representative(reduced=False)

connected_components()

Lists the connected components of the Stratum.

OUTPUT:

list – a list of connected components of stratum

EXAMPLES:
sage: AbelianStratum(0).connected_components()
[H_hyp(0)]

sage: AbelianStratum(2).connected_components()
[H_hyp(2)]
sage: AbelianStratum(1,1).connected_components()
[H_hyp(1, 1)]

sage: AbelianStratum(4).connected_components()
[H_hyp(4), H_odd(4)]
sage: AbelianStratum(3,1).connected_components()
[H_c(3, 1)]
sage: AbelianStratum(2,2).connected_components()
[H_hyp(2, 2), H_odd(2, 2)]
sage: AbelianStratum(2,1,1).connected_components()
[H_c(2, 1, 1)]
sage: AbelianStratum(1,1,1,1).connected_components()
[H_c(1, 1, 1, 1)]

genus()
Returns the genus of the stratum.

OUTPUT:
integer – the genus

EXAMPLES:

sage: AbelianStratum(0).genus()
1
sage: AbelianStratum(1,1).genus()
2
sage: AbelianStratum(3,2,1).genus()
4

is_connected()
Tests if the strata is connected.

OUTPUT:
boolean – True if it is connected else False

EXAMPLES:

sage: AbelianStratum(2).is_connected()
True
sage: AbelianStratum(2).connected_components()
[H_hyp(2)]

sage: AbelianStratum(2,2).is_connected()
False
sage: AbelianStratum(2,2).connected_components()
[H_hyp(2, 2), H_odd(2, 2)]

nintervals()
Returns the number of intervals of any iet of the strata.

OUTPUT:
integer – the number of intervals for any associated iet

EXAMPLES:

```python
sage: AbelianStratum(0).nintervals()
2
sage: AbelianStratum(0,0).nintervals()
3
sage: AbelianStratum(2).nintervals()
4
sage: AbelianStratum(1,1).nintervals()
5
```

sage.dynamics.flat_surfaces.strata.CCA
alias of sage.dynamics.flat_surfaces.strata.ConnectedComponentOfAbelianStratum

class sage.dynamics.flat_surfaces.strata.ConnectedComponentOfAbelianStratum(parent)
Bases: sage.structure.sage_object.SageObject

Connected component of Abelian stratum.

**Warning:** Internal class! Do not use directly!

```python
sage: a = AbelianStratum(2,4,0,marked_separatrix='out')
sage: a_odd, a_even = a.connected_components()
sage: a_odd.representative().attached_out_degree()
2
sage: a_even.representative().attached_out_degree()
2
```

```python
sage: a = AbelianStratum(0,4,2,marked_separatrix='out')
sage: a_odd, a_even = a.connected_components()
sage: a_odd.representative().attached_out_degree()
0
sage: a_even.representative().attached_out_degree()
0
```

```python
sage: a = AbelianStratum(3,2,1,marked_separatrix='out')
sage: a_c = a.connected_components()[0]
sage: a_c.representative().attached_out_degree()
3
```

```python
sage: a = AbelianStratum(2,3,1,marked_separatrix='out')
sage: a_c = a.connected_components()[0]
sage: a_c.representative().attached_out_degree()
2
```

```python
sage: a = AbelianStratum(1,3,2,marked_separatrix='out')
sage: a_c = a.connected_components()[0]
sage: a_c.representative().attached_out_degree()
1
```

Tests for incoming separatrices:
sage: a = AbelianStratum(4,2,0,marked_separatrix='in')
sage: a_odd, a_even = a.connected_components()
sage: a_odd.representative().attached_in_degree()
4
sage: a_even.representative().attached_in_degree()
4

sage: a = AbelianStratum(2,4,0,marked_separatrix='in')
sage: a_odd, a_even = a.connected_components()
sage: a_odd.representative().attached_in_degree()
2
sage: a_even.representative().attached_in_degree()
2

sage: a = AbelianStratum(0,4,2,marked_separatrix='in')
sage: a_odd, a_even = a.connected_components()
sage: a_odd.representative().attached_in_degree()
0
sage: a_even.representative().attached_in_degree()
0

sage: a = AbelianStratum(3,2,1,marked_separatrix='in')
sage: a_c = a.connected_components()[0]
sage: a_c.representative().attached_in_degree()
3

sage: a = AbelianStratum(2,3,1,marked_separatrix='in')
sage: a_c = a.connected_components()[0]
sage: a_c.representative().attached_in_degree()
2

sage: a = AbelianStratum(1,3,2,marked_separatrix='in')
sage: a_c = a.connected_components()[0]
sage: a_c.representative().attached_in_degree()
1

**genus()**

Returns the genus of the surfaces in this connected component.

**OUTPUT:**

integer – the genus of the surface

**EXAMPLES:**

sage: a = AbelianStratum(6,4,2,0,0)

sage: c_odd, c_even = a.connected_components()

sage: c_odd.genus()
7

sage: c_even.genus()
7

sage: a = AbelianStratum([[1]*8])

sage: c = a.connected_components()[0]

sage: c.genus()
5

3.1. Strata of differentials on Riemann surfaces
nintervals()  
Returns the number of intervals of the representative.  

OUTPUT:  
integer – the number of intervals in any representative  

EXAMPLES:

```
sage: a = AbelianStratum(6,4,2,0,0)
sage: c_odd, c_even = a.connected_components()
sage: c_odd.nintervals()
18
sage: c_even.nintervals()
18
```

```
sage: a = AbelianStratum([1]*8)
sage: c = a.connected_components()[0]
sage: c.nintervals()
17
```

parent()  
The stratum of this component  

OUTPUT:  
stratum - the stratum where this component leaves  

EXAMPLES:

```
sage: p = iet.Permutation('a b','b a')
doctest:warning
...
DeprecationWarning: Permutation is deprecated and will be removed from Sage.
You are advised to install the surface_dynamics package via:
sage -pip install surface_dynamics
If you do not have write access to the Sage installation you can alternatively do
sage -pip install surface_dynamics --user
The package surface_dynamics subsumes all flat surface related computation that are currently available in Sage. See more information at
http://www.labri.fr/perso/vdelecro/surface-dynamics/latest/
See http://trac.sagemath.org/20695 for details.
sage: c = p.connected_component()
sage: c.parent()
H(0)
```

rauzy_diagram(reduced=True)  
Returns the Rauzy diagram associated to this connected component.  

OUTPUT:  
rauzy diagram – the Rauzy diagram associated to this stratum  

EXAMPLES:

```
sage: c = AbelianStratum(0).connected_components()[0]
sage: r = c.rauzy_diagram()
```
representative \((\text{reduced}=\text{True}, \text{alphabet}=\text{None})\)

Returns the Zorich representative of this connected component.

Zorich constructs explicitly interval exchange transformations for each stratum in [Zor2008].

INPUT:

- \text{reduced} - boolean (default: True): whether you obtain a reduced or labelled permutation
- \text{alphabet} - an alphabet or None: whether you want to specify an alphabet for your permutation

OUTPUT:

permutation – a permutation which lives in this component

EXAMPLES:

```
sage: c = AbelianStratum(1,1,1,1).connected_components()[0]
sage: c
H_{c}(1, 1, 1, 1)
sage: p = c.representative(alphabet=range(9))
sage: p
0 1 2 3 4 5 6 7 8
4 3 2 5 8 7 6 1 0
```

Warning: Internal class! Do not use directly!

representative \((\text{reduced}=\text{True}, \text{alphabet}=\text{None})\)

Returns the Zorich representative of this connected component.

Zorich constructs explicitly interval exchange transformations for each stratum in [Zor2008].

EXAMPLES:

```
sage: c = AbelianStratum(6).connected_components()[2]
sage: c
H_{c}(6)
sage: p = c.representative(alphabet=range(8))
sage: p
0 1 2 3 4 5 6 7
5 4 3 2 7 6 1 0
```

(continues on next page)
sage.dynamics.flat_surfaces.strata.HypCCA
alias of sage.dynamics.flat_surfaces.strata.HypConnectedComponentOfAbelianStratum
class sage.dynamics.flat_surfaces.strata.HypConnectedComponentOfAbelianStratum

Hyperelliptic component of Abelian stratum.

Warning: Internal class! Do not use directly!

representative (reduced=True, alphabet=None)

Returns the Zorich representative of this connected component.

Zorich constructs explicitly interval exchange transformations for each stratum in [Zor2008].

INPUT:

• reduced - boolean (default: True): whether you obtain a reduced or labelled permutation

• alphabet - alphabet or None (default: None): whether you want to specify an alphabet for your representative

EXAMPLES:

```python
sage: c = AbelianStratum(0).connected_components()[0]
sage: c
H_hyp(0)
sage: p = c.representative(alphabet="01")
sage: p
0 1
1 0
sage: p.connected_component()
H_hyp(0)
```

```python
sage: c = AbelianStratum(0,0).connected_components()[0]
sage: c
H_hyp(0, 0)
sage: p = c.representative(alphabet="abc")
sage: p
a b c
c b a
sage: p.connected_component()
H_hyp(0, 0)
```

```python
sage: c = AbelianStratum(2).connected_components()[0]
sage: c
H_hyp(2)
sage: p = c.representative(alphabet="ABCD")
sage: p
A B C D
D C B A
```
sage: p.connected_component()
H_hyp(2)

sage: c = AbelianStratum(1,1).connected_components()[0]
sage: c
H_hyp(1, 1)

sage: p = c.representative(alphabet="01234")

sage: p
0 1 2 3 4
4 3 2 1 0

sage: p.connected_component()
H_hyp(1, 1)

sage.dynamics.flat_surfaces.strata.
NonHypCCA
alias of sage.dynamics.flat_surfaces.strata.NonHypConnectedComponentOfAbelianStratum
class sage.dynamics.flat_surfaces.strata.
NonHypConnectedComponentOfAbelianStratum (parent)
Bases: sage.dynamics.flat_surfaces.strata.ConnectedComponentOfAbelianStratum

Non hyperelliptic component of Abelian stratum.

Warning: Internal class! Do not use directly!

sage.dynamics.flat_surfaces.strata.
OddCCA
alias of sage.dynamics.flat_surfaces.strata.OddConnectedComponentOfAbelianStratum (parent)
class sage.dynamics.flat_surfaces.strata.
OddConnectedComponentOfAbelianStratum (parent)
Bases: sage.dynamics.flat_surfaces.strata.
ConnectedComponentOfAbelianStratum

Connected component of an Abelian stratum with odd spin parity.

Warning: Internal class! Do not use directly!

representative (reduced=True, alphabet=None)
Returns the Zorich representative of this connected component.
Zorich constructs explicitly interval exchange transformations for each stratum in [Zor2008].

EXAMPLES:

sage: a = AbelianStratum(6).connected_components()[1]
sage: a.representative(alphabet=range(8))
0 1 2 3 4 5 6 7
3 2 5 4 7 6 1 0

sage: a = AbelianStratum(4,4).connected_components()[1]
sage: a.representative(alphabet=range(11))
0 1 2 3 4 5 6 7 8 9 10
3 2 5 4 6 8 7 10 9 1 0

3.1. Strata of differentials on Riemann surfaces
3.2 Strata of quadratic differentials on Riemann surfaces

Warning: This module is deprecated. You are advised to install and use the surface_dynamics package instead available at https://pypi.python.org/pypi/surface_dynamics/

class sage.dynamics.flat_surfaces.quadratic_strata.QuadraticStratum(*l)
   Bases: sage.structure.sage_object.SageObject

   Stratum of quadratic differentials.

   genus()
      Returns the genus.

   EXAMPLES:

     sage: QuadraticStratum(-1,-1,-1,-1).genus()
     0
4.1 Class factories for Interval exchange transformations.

Warning: This module is deprecated. You are advised to install and use the surface_dynamics package instead available at https://pypi.python.org/pypi/surface_dynamics/

This library is designed for the usage and manipulation of interval exchange transformations and linear involutions. It defines specialized types of permutation (constructed using iet.Permutation()) some associated graph (constructed using iet.RauzyGraph()) and some maps of intervals (constructed using iet.IntervalExchangeTransformation()).

EXAMPLES:

Creation of an interval exchange transformation:

```
sage: T = iet.IntervalExchangeTransformation(('a b', 'b a'), (sqrt(2), 1))
doctest:warning
...
DeprecationWarning: IntervalExchangeTransformation is deprecated and will be removed from Sage.
You are advised to install the surface_dynamics package via:
sage -pip install surface_dynamics
If you do not have write access to the Sage installation you can alternatively do
    sage -pip install surface_dynamics --user
The package surface_dynamics subsumes all flat surface related computation that are currently available in Sage. See more information at 
    http://www.labri.fr/perso/vdelecro/surface-dynamics/latest/
See http://trac.sagemath.org/20695 for details.
doctest:warning
...
DeprecationWarning: Permutation is deprecated and will be removed from Sage.
You are advised to install the surface_dynamics package via:
sage -pip install surface_dynamics
If you do not have write access to the Sage installation you can alternatively do
    sage -pip install surface_dynamics --user
The package surface_dynamics subsumes all flat surface related computation that are currently available in Sage. See more
```
It can also be initialized using permutation (group theoretic ones):

```sage
sage: p = Permutation([3,2,1])
sage: T = iet.IntervalExchangeTransformation(p, [1/3,2/3,1])
sage: T
Interval exchange transformation of [0, 2] with permutation
1 2 3
3 2 1
```

For the manipulation of permutations of iet, there are special types provided by this module. All of them can be constructed using the constructor `iet.Permutation`. For the creation of labelled permutations of interval exchange transformation:

```sage
sage: pl = iet.Permutation('a b c', 'c b a')
sage: pl
a b c
c b a
```

They can be used for initialization of an iet:

```sage
sage: p = iet.Permutation('a b', 'b a')
sage: T = iet.IntervalExchangeTransformation(p, [1,sqrt(2)])
sage: T
Interval exchange transformation of [0, sqrt(2) + 1] with permutation
a b
b a
```

You can also, create labelled permutations of linear involutions:

```sage
doctest:warning
DeprecationWarning: GeneralizedPermutation is deprecated and will be removed from Sage.
You are advised to install the surface_dynamics package via:
sage -pip install surface_dynamics
If you do not have write access to the Sage installation you can alternatively do
sage -pip install surface_dynamics --user
The package surface_dynamics subsumes all flat surface related computation that are currently available in Sage. See more information at
http://www.labri.fr/perso/vdelecro/surface-dynamics/latest/
See http://trac.sagemath.org/20695 for details.
sage: p
a a b
b c c
```

Sometimes it’s more easy to deal with reduced permutations:
Permutations:

```python
sage: p = iet.Permutation('a b c', 'c b a', reduced = True)
sage: p
a b c
c b a
```

Permutations with flips:

```python
sage: p1 = iet.Permutation('a b c', 'c b a', flips = ['a','c'])
sage: p1
-a b -c
-c b -a
```

Creation of Rauzy diagrams:

```python
sage: r = iet.RauzyDiagram('a b c', 'c b a')
```

```
doctest:warning
...
DeprecationWarning: RauzyDiagram is deprecated and will be removed from Sage.
You are advised to install the surface_dynamics package via:
    sage -pip install surface_dynamics
If you do not have write access to the Sage installation you can alternatively do
    sage -pip install surface_dynamics --user
The package surface_dynamics subsumes all flat surface related computation that are currently available in Sage. See more information at
    http://www.labri.fr/perso/vdelecro/surface-dynamics/latest/
See http://trac.sagemath.org/20695 for details.
```

Reduced Rauzy diagrams are constructed using the same arguments than for permutations:

```python
sage: r = iet.RauzyDiagram('a b c', 'c b a')
sage: r_red = iet.RauzyDiagram('a b c', 'c b a', reduced=True)
sage: r.cardinality()
12
sage: r_red.cardinality()
4
```

By default, Rauzy diagrams are generated by induction on the right. You can use several options to enlarge (or restrict) the diagram (try help(iet.RauzyDiagram) for more precisions):

```python
sage: r1 = iet.RauzyDiagram('a b c', 'c b a', right_induction=True)
sage: r2 = iet.RauzyDiagram('a b c', 'c b a', left_right_inversion=True)
```

You can consider self similar iet using path in Rauzy diagrams and eigenvectors of the corresponding matrix:

```python
sage: p = iet.Permutation("a b c d", "d c b a")
sage: d = p.rauzy_diagram()
sage: g = d.path(p, 't', 't', 'b', 't', 'b', 'b', 't', 'b')
sage: g
Path of length 8 in a Rauzy diagram
sage: g.is_loop()
True
sage: g.is_full()
True
sage: m = g.matrix()
sage: v = m.eigenvectors_right()[-1][1][0]
```

(continues on next page)
sage: T1 = iet.IntervalExchangeTransformation(p, v)
sage: T2 = T1.rauzy_move(iterations=8)
sage: T1.normalize(1) == T2.normalize(1)
True

REFERENCES:
• [BL2008]
• [DN1990]
• [Nog1985]
• [Rau1979]
• [Vee1978]
• [Zor]

AUTHORS:
• Vincent Delecroix (2009-09-29): initial version

sage.dynamics.interval_exchanges.constructors.GeneralizedPermutation(*args, **kargs)

Returns a permutation of an interval exchange transformation.

Those permutations are the combinatoric part of linear involutions and were introduced by Danthony-Nogueira [DN1990]. The full combinatoric study and precise links with strata of quadratic differentials was achieved few years later by Boissy-Lanneau [BL2008].

INPUT:
• intervals - strings, list, tuples
• reduced - boolean (default: False) specifies reduction. False means labelled permutation and True means reduced permutation.
• flips - iterable (default: None) the letters which correspond to flipped intervals.

OUTPUT:
generalized permutation – the output type depends on the data.

EXAMPLES:
Creation of labelled generalized permutations:

sage: iet.GeneralizedPermutation('a b b','c c a')

a b b

sage: iet.GeneralizedPermutation('a a','b b c c')

a a

b b c c

sage: iet.GeneralizedPermutation([[0,1,2,3,1],[4,2,5,3,5,4,0]])

0 1 2 3 1

4 2 5 3 5 4 0

Creation of reduced generalized permutations:

sage: iet.GeneralizedPermutation('a b b', 'c c a', reduced = True)

a b b

c c a
sage: iet.GeneralizedPermutation('a a b b', 'c c d d', reduced = True)
a a b b
c c d d

Creation of flipped generalized permutations:

sage: iet.GeneralizedPermutation('a b c a', 'd c d b', flips = ['a','b'])
-a -b c -a
d c d -b

sage.dynamics.interval_exchanges.constructors.IET(permutation=None, lengths=None)
Constructs an Interval exchange transformation.

An interval exchange transformation (or iet) is a map from an interval to itself. It is defined on the interval except at a finite number of points (the singularities) and is a translation on each connected component of the complement of the singularities. Moreover it is a bijection on its image (or it is injective).

An interval exchange transformation is encoded by two datas. A permutation (that corresponds to the way we exchange the intervals) and a vector of positive reals (that corresponds to the lengths of the complement of the singularities).

INPUT:

• permutation - a permutation
• lengths - a list or a dictionary of lengths

OUTPUT:

interval exchange transformation – an map of an interval

EXAMPLES:

Two initialization methods, the first using a iet.Permutation:

sage: p = iet.Permutation('a b c','c b a')
sage: t = iet.IntervalExchangeTransformation(p, {'a':1,'b':0.4523,'c':2.8})

The second is more direct:

sage: t = iet.IntervalExchangeTransformation(('a b','b a'),{'a':1,'b':4})

It’s also possible to initialize the lengths only with a list:

sage: t = iet.IntervalExchangeTransformation(('a b c','c b a'),[0.123,0.4,2])

The two fundamental operations are Rauzy move and normalization:

sage: t = iet.IntervalExchangeTransformation(('a b c','c b a'),[0.123,0.4,2])
sage: s = t.rauzy_move()
sage: s_n = s.normalize(t.length())
sage: s_n.length() == t.length()
True

A not too simple example of a self similar interval exchange transformation:

sage: p = iet.Permutation('a b c d','d c b a')
sage: d = p.rauzy_diagram()
sage: g = d.path(p, 't', 't', 'b', 't', 'b', 'b', 't', 'b')

(continues on next page)
sage: m = g.matrix()
sage: v = m.eigenvectors_right()[-1][1][0]
sage: t = iet.IntervalExchangeTransformation(p,v)
sage: s = t.rauzy_move(iterations=8)
sage: s.normalize() == t.normalize()
True

sage.dynamics.interval_exchanges.constructors.IntervalExchangeTransformation(permuation=None, lengths=None)

Constructs an Interval exchange transformation.

An interval exchange transformation (or iet) is a map from an interval to itself. It is defined on the interval except at a finite number of points (the singularities) and is a translation on each connected component of the complement of the singularities. Moreover it is a bijection on its image (or it is injective).

An interval exchange transformation is encoded by two datas. A permutation (that corresponds to the way we exchange the intervals) and a vector of positive reals (that corresponds to the lengths of the complement of the singularities).

INPUT:

* permutation - a permutation
* lengths - a list or a dictionary of lengths

OUTPUT:

interval exchange transformation – an map of an interval

EXAMPLES:

Two initialization methods, the first using a iet.Permutation:

```python
sage: p = iet.Permutation('a b c','c b a')
sage: t = iet.IntervalExchangeTransformation(p, {'a':1,'b':0.4523,'c':2.8})
```

The second is more direct:

```python
sage: t = iet.IntervalExchangeTransformation(('a b','b a'),{'a':1,'b':4})
```

It's also possible to initialize the lengths only with a list:

```python
sage: t = iet.IntervalExchangeTransformation(('a b c','c b a'),[0.123,0.4,2])
```

The two fundamental operations are Rauzy move and normalization:

```python
sage: t = iet.IntervalExchangeTransformation(('a b c','c b a'),[0.123,0.4,2])
sage: s = t.rauzy_move()
sage: s_n = s.normalize(t.length())
sage: s_n.length() == t.length()
True
```

A not too simple example of a self similar interval exchange transformation:

```python
sage: p = iet.Permutation('a b c d','d c b a')
sage: d = p.rauzy_diagram()
sage: g = d.path(p, 't', 't', 'b', 't', 'b', 'b', 't', 'b')
sage: m = g.matrix()
sage: v = m.eigenvectors_right()[-1][1][0]
```
sage: t = iet.IntervalExchangeTransformation(p,v)
sage: s = t.rauzy_move(iterations=8)
sage: s.normalize() == t.normalize()
True

sage.dynamics.interval_exchanges.constructors.Permutation(*args,**kargs)
Returns a permutation of an interval exchange transformation.

Those permutations are the combinatoric part of an interval exchange transformation (IET). The combinatorial study of those objects starts with Gerard Rauzy [Rau1979] and William Veech [Vee1978].

The combinatoric part of interval exchange transformation can be taken independently from its dynamical origin. It has an important link with strata of Abelian differential (see strata)

INPUT:

• intervals - string, two strings, list, tuples that can be converted to two lists
• reduced - boolean (default: False) specifies reduction. False means labelled permutation and True means reduced permutation.
• flips - iterable (default: None) the letters which correspond to flipped intervals.

OUTPUT:

permutation – the output type depends of the data.

EXAMPLES:

Creation of labelled permutations

sage: iet.Permutation('a b c d', 'd c b a')
 a b c d
 d c b a

sage: iet.Permutation([[0,1,2,3],[2,1,3,0]])
 0 1 2 3
 2 1 3 0

sage: iet.Permutation([0, 'A', 'B', 1], ['B', 0, 1, 'A'])
 0 A B 1
 B 0 1 A

Creation of reduced permutations:

sage: iet.Permutation('a b c', 'c b a', reduced = True)
 a b c
 c b a

sage: iet.Permutation([0, 1, 2, 3], [1, 3, 0, 2])
 0 1 2 3
 1 3 0 2

Creation of flipped permutations:

sage: iet.Permutation('a b c', 'c b a', flips=['a','b'])
 -a -b c
 c -b -a

sage: iet.Permutation('a b c', 'c b a', flips=['a'], reduced=True)
 -a b c
 c b -a

4.1. Class factories for Interval exchange transformations.
sage: p = iet.Permutation('a b c','c b a')
sage: iet.Permutation(p) == p
True
sage: iet.Permutation(p, reduced=True) == p.reduced()
True

sage: p = iet.Permutation('a','a',flips='a',reduced=True)
sage: iet.Permutation(p) == p
True

sage: p = iet.Permutation('a b c','c b a',flips='a')
sage: iet.Permutation(p) == p
True
sage: iet.Permutation(p, reduced=True) == p.reduced()
True

sage: p = iet.Permutation('a b c','c b a',reduced=True)
sage: iet.Permutation(p) == p
True

sage: iet.Permutation('a b c','c b a',reduced='badly')
Traceback (most recent call last):
  ... *
TypeError: reduced must be of type boolean

sage: iet.Permutation('a','a',flips='b',reduced=True)
Traceback (most recent call last):
  ... *
ValueError: flips contains not valid letters

sage: iet.Permutation('a b c','c a a',reduced=True)
Traceback (most recent call last):
  ... *
ValueError: letters must appear once in each interval

sage.dynamics.interval_exchanges.constructors.Permutations_iterator(nintervals=None, irreducible=True, reduced=False, alphabet=None)

Returns an iterator over permutations.

This iterator allows you to iterate over permutations with given constraints. If you want to iterate over permutations coming from a given stratum you have to use the module strata and generate Rauzy diagrams from connected components.

INPUT:

- nintervals - non negative integer
- irreducible - boolean (default: True)
- reduced - boolean (default: False)
- alphabet - alphabet (default: None)

OUTPUT:

iterator – an iterator over permutations
EXAMPLES:

Generates all reduced permutations with given number of intervals:

```python
sage: P = iet.Permutations_iterator(nintervals=2,alphabet="ab",reduced=True)
```
```doctest:warning
...
DeprecationWarning: iet_Permutations_iterator is deprecated and will be removed_...
\rightarrow from Sage.
```
You are advised to install the surface_dynamics package via:
```
sage -pip install surface_dynamics
```
If you do not have write access to the Sage installation you can alternatively do
```
sage -pip install surface_dynamics --user
```
The package surface_dynamics subsumes all flat surface related computation that are currently available in Sage. See more information at
```
http://www.labri.fr/perso/vdelecro/surface-dynamics/latest/
```
See http://trac.sagemath.org/20695 for details.
```sage: for p in P:
......: print(p)
......: print(" * *
```
a b
b a
* *
sage: P = iet.Permutations_iterator(nintervals=3,alphabet="abc",reduced=True)
```sage: for p in P:
......: print(p)
......: print(" * *
```
a b c
b c a
* * *
a b c
c a b
* *
a b c
c b a
* * *
```

```
sage.dynamics.interval_exchanges.constructors.RauzyDiagram(*args, **kargs)
```
Return an object coding a Rauzy diagram.

The Rauzy diagram is an oriented graph with labelled edges. The set of vertices corresponds to the permutations obtained by different operations (mainly the .rauzy_move() operations that corresponds to an induction of interval exchange transformation). The edges correspond to the action of the different operations considered.

It first appeared in the original article of Rauzy [Rau1979].

INPUT:

- `intervals` - lists, or strings, or tuples
- `reduced` - boolean (default: False) to precise reduction
- `flips` - list (default: []) for flipped permutations
- `right_induction` - boolean (default: True) consideration of left induction in the diagram
- `left_induction` - boolean (default: False) consideration of right induction in the diagram
- `left_right_inversion` - boolean (default: False) consideration of inversion

4.1. Class factories for Interval exchange transformations.
• **top_bottom_inversion** - boolean (default: False) consideration of reversion

• **symmetric** - boolean (default: False) consideration of the symmetric operation

**OUTPUT:**

Rauzy diagram – the Rauzy diagram that corresponds to your request

**EXAMPLES:**

Standard Rauzy diagrams:

```
sage: iet.RauzyDiagram('a b c d', 'd b c a')
Rauzy diagram with 12 permutations
sage: iet.RauzyDiagram('a b c d', 'd b c a', reduced = True)
Rauzy diagram with 6 permutations
```

Extended Rauzy diagrams:

```
sage: iet.RauzyDiagram('a b c d', 'd b c a', symmetric=True)
Rauzy diagram with 144 permutations
```

Using Rauzy diagrams and path in Rauzy diagrams:

```
sage: r = iet.RauzyDiagram('a b c', 'c b a')
sage: r
Rauzy diagram with 3 permutations
sage: p = iet.Permutation('a b c','c b a')
sage: p in r
True
sage: g0 = r.path(p, 'top', 'bottom','top')
sage: g1 = r.path(p, 'bottom', 'top', 'bottom')
sage: g0.is_loop(), g1.is_loop()
(True, True)
sage: g0.is_full(), g1.is_full()
(False, False)
sage: g = g0 + g1
sage: g
Path of length 6 in a Rauzy diagram
sage: g.is_loop(), g.is_full()
(True, True)
sage: m = g.matrix()
sage: m
[1 1 1]
[2 4 1]
[2 3 2]
sage: s = g.orbit_substitution()
sage: s
WordMorphism: a->acbbc, b->acbbcbcb, c->acbc
sage: s.incidence_matrix() == m
True
```

We can then create the corresponding interval exchange transformation and comparing the orbit of 0 to the fixed point of the orbit substitution:

```
sage: v = m.eigenvectors_right()[-1][1][0]
sage: T = iet.IntervalExchangeTransformation(p, v).normalize()
sage: T
Interval exchange transformation of [0, 1[ with permutation
a b c
```

(continues on next page)
4.2 Labelled permutations

A labelled (generalized) permutation is better suited to study the dynamic of a translation surface than a reduced one (see the module `sage.dynamics.interval_exchanges.reduced`). The latter is more adapted to the study of strata. This kind of permutation was introduced by Yoccoz [Yoc2005] (see also [MMY2003]).

In fact, there is a geometric counterpart of labelled permutations. They correspond to translation surfaces with marked outgoing separatrices (i.e. we fix a label for each of them).

Remarks that Rauzy diagram of reduced objects are significantly smaller than the one for labelled object (for the permutation a b d b e / e d c a c the labelled Rauzy diagram contains 8760 permutations, and the reduced only 73). But, as it is in geometrical way, the labelled Rauzy diagram is a covering of the reduced Rauzy diagram.

AUTHORS:

• Vincent Delecroix (2009-09-29) : initial version

```
c b a
sage: w1 = []
sage: x = 0
sage: for i in range(20):
    ....:     w1.append(T.in_which_interval(x))
    ....:     x = T(x)
sage: w1 = Word(w1)
sage: w1
word: acbbcacbcacbbcbbcacb
sage: w2 = s.fixed_point('a')
sage: w2[:20]
word: acbbcacbcacbbcbbcacb
sage: w2[:20] == w1
True
```

```python
4.2. Labelled permutations 51
```

**Warning:** This module is deprecated. You are advised to install and use the surface_dynamics package instead available at https://pypi.python.org/pypi/surface_dynamics/

```python
class sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutation(
    intervals=None,
    alphabet=None,
    flips=None
):
    Bases: sage.dynamics.interval_exchanges.labelled.LabelledPermutation

    General template for labelled objects

    **Warning:** Internal class! Do not use directly!

    list (flips=False)
    Returns a list associated to the permutation.

    INPUT:
        • flips - boolean (default: False)
```
OUTPUT:

list – two lists of labels

EXAMPLES:

```python
sage: p = iet.GeneralizedPermutation('0 0 1 2 2 1', '3 3', flips='1')
sage: p.list(flips=True)
[[(0', 1), ('0', 1), ('1', -1), ('2', 1), ('2', 1), ('1', -1)], [(3', 1), (3', 1)]]
sage: p.list(flips=False)
[['0', '0', '1', '2', '2', '1'], ['3', '3']]
```

The list can be used to reconstruct the permutation

```python
sage: p = iet.Permutation('a b c', 'c b a', flips='a')
sage: p == iet.Permutation(p.list(), flips=p.flips())
True
```  
```python
sage: p = iet.GeneralizedPermutation('a b b c', 'c d d a', flips='ad')
sage: p == iet.GeneralizedPermutation(p.list(), flips=p.flips())
True
```

```python
class sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutationIET(intervals=None, alphabet=None, flips=None)

Bases: sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutation, sage.dynamics.interval_exchanges.template.FlippedPermutationIET, sage.dynamics.interval_exchanges.labelled.LabelledPermutationIET

Flipped labelled permutation from iet.

EXAMPLES:

Reducibility testing (does not depends of flips):

```python
sage: p = iet.Permutation('a b c', 'c b a', flips='a')
sage: p.is_irreducible()
True
sage: q = iet.Permutation('a b c d', 'b a d c', flips='bc')
sage: q.is_irreducible()
False
```

Rauzy movability and Rauzy move:

```python
sage: p = iet.Permutation('a b c', 'c b a', flips='a')
sage: p
-a b c
c b -a
sage: p.rauzy_move(1)
-c -a b
-c b -a
sage: p.rauzy_move(0)
-a b c
c -a b
```

Rauzy diagrams:
AUTHORS:

• Vincent Delecroix (2009-09-29): initial version

rauzy_diagram(**kargs)

Returns the Rauzy diagram associated to this permutation.

For more information, try help(iet.RauzyDiagram)

OUTPUT:

RauzyDiagram – the Rauzy diagram of self

EXAMPLES:

```python
sage: p = iet.Permutation('a b c', 'c b a', flips='a')
sage: p.rauzy_diagram()
Rauzy diagram with 3 permutations
```

rauzy_move(winner=None, side=None)

Returns the Rauzy move.

INPUT:

• winner - ‘top’ (or ‘t’ or 0) or ‘bottom’ (or ‘b’ or 1)
• side - (default: ‘right’) ‘right’ (or ‘r’) or ‘left’ (or ‘l’)

OUTPUT:

permutation – the Rauzy move of self

EXAMPLES:

```python
sage: p = iet.Permutation('a b', 'b a', flips='a')
sage: p.rauzy_move('top')
-a b
 b -a
sage: p.rauzy_move('bottom')
-b -a
-b -a
```

```python
sage: p = iet.Permutation('a b c', 'c b a', flips='b')
sage: p.rauzy_move('top')
a -b c
 c a -b
```
reduced()
The associated reduced permutation.

OUTPUT:

permutation – the associated reduced permutation

EXAMPLES:

```python
sage: p = iet.Permutation('a b c', 'c b a', flips='a')
sage: q = iet.Permutation('a b c', 'c b a', flips='a', reduced=True)
sage: p.reduced() == q
True
```

class sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutationLI

Bases: sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutation,
sage.dynamics.interval_exchanges.template.FlippedPermutationLI,
sage.dynamics.interval_exchanges.labelled.LabelledPermutationLI

Flipped labelled quadratic (or generalized) permutation.

EXAMPLES:

Reducibility testing:

```python
sage: p = iet.GeneralizedPermutation('a b b', 'c c a', flips='a')
sage: p.is_irreducible()
True
```

Reducibility testing with associated decomposition:

```python
sage: p = iet.GeneralizedPermutation('a b c a', 'b d d c', flips='ab')
sage: p.is_irreducible()
False
sage: test, decomp = p.is_irreducible(return_decomposition = True)
sage: test
False
sage: decomp
([['a']], ['c', 'a'], [], ['c'])
```

Rauzy movability and Rauzy move:

```python
sage: p = iet.GeneralizedPermutation('a a b b c c', 'd', flips='d')
sage: p.has_rauzy_move(0)
False
sage: p.has_rauzy_move(1)
True
sage: p = iet.GeneralizedPermutation('a a b', 'b c c', flips='c')
sage: p.has_rauzy_move(0)
True
```
left_rauzy_move(winner)

Perform a Rauzy move on the left.

**INPUT:**
- winner - either 'top' or 'bottom' ('t' or 'b' for short)

**OUTPUT:**
- a permutation

**EXAMPLES:**

```python
sage: p = iet.GeneralizedPermutation('a a b', 'b c c')
sage: p.left_rauzy_move(0)
a a b b
c c
sage: p.left_rauzy_move(1)
a a b c c
```

rauzy_diagram(**kargs)

Returns the associated Rauzy diagram.

For more information, try help(RauzyDiagram)

**OUTPUT:**
- a RauzyDiagram

**EXAMPLES:**

```python
sage: p = iet.GeneralizedPermutation('a b b', 'c c a')
sage: p.rauzy_diagram()
```

reduced()

The associated reduced permutation.

**OUTPUT:**
- the associated reduced permutation

**EXAMPLES:**

```python
sage: p = iet.GeneralizedPermutation('a a', 'b b c c', flips='a')
sage: q = iet.GeneralizedPermutation('a a', 'b b c c', flips='a', reduced=True)
sage: p.reduced() == q
True
```
**right_rauzy_move** *(winner)*

Perform a Rauzy move on the right (the standard one).

**INPUT:**

- `winner` - either 'top' or 'bottom' ('t' or 'b' for short)

**OUTPUT:**

permutation – the Rauzy move of `self`

**EXAMPLES:**

```python
sage: p = iet.GeneralizedPermutation('a a b','b c c',flips='c')
sage: p.right_rauzy_move(0)
a a b
-c b -c
sage: p.right_rauzy_move(1)
a a
-b -c -b -c
```

```python
sage: p = iet.GeneralizedPermutation('a b b','c c a',flips='ab')
sage: p.right_rauzy_move(0)
a -b a -b
c c
sage: p.right_rauzy_move(1)
b -a b
c c -a
```
sage: p = iet.Permutation('a b c d','c d b a')
sage: p.erase_letter('a')
b c d
c d b
sage: p.erase_letter('b')
a c d
c d a
sage: p.erase_letter('c')
   a b d
d b a
sage: p.erase_letter('d')
   a b c
c b a

sage: p = iet.GeneralizedPermutation('a b b','c c a')
sage: p.erase_letter('a')
b b
   c c

Beware, there is no validity check for permutation from linear involutions:

sage: p = iet.GeneralizedPermutation('a b b','c c a')
sage: p.erase_letter('b')
a
   c c a

length

Returns a 2-uple of lengths.

p.length() is identical to (p.length_top(), p.length_bottom()) If an interval is specified, it returns the length of the specified interval.

INPUT:

• interval - None, ‘top’ or ‘bottom’

OUTPUT:

tuple – a 2-uple of integers

EXAMPLES:

sage: iet.Permutation('a b c','c b a').length()
   (3, 3)
sage: iet.GeneralizedPermutation('a a','b b c c').length()
   (2, 4)
sage: iet.GeneralizedPermutation('a a b b','c c').length()
   (4, 2)

length_bottom

Returns the number of intervals in the bottom segment.

OUTPUT:

integer – number of intervals

EXAMPLES:

sage: iet.Permutation('a b','b a').length_bottom()
   2

(continues on next page)
sage: iet.GeneralizedPermutation('a a','b b c c').length_bottom()
4
sage: iet.GeneralizedPermutation('a a b b','c c').length_bottom()
2

**length_top()**

Returns the number of intervals in the top segment.

**OUTPUT:**

integer – number of intervals

**EXAMPLES:**

sage: iet.Permutation('a b c','c b a').length_top()
3
sage: iet.GeneralizedPermutation('a a','b b c c').length_top()
doctest:warning
...
DeprecationWarning: GeneralizedPermutation is deprecated and will be removed from Sage.
You are advised to install the surface_dynamics package via:
  sage -pip install surface_dynamics
If you do not have write access to the Sage installation you can alternatively do
  sage -pip install surface_dynamics --user
The package surface_dynamics subsumes all flat surface related computation that are currently available in Sage. See more information at
  http://www.labri.fr/perso/vdelecro/surface-dynamics/latest/
See http://trac.sagemath.org/20695 for details.

sage: iet.GeneralizedPermutation('a a b b','c c').length_top()
4

**list()**

Returns a list of two lists corresponding to the intervals.

**OUTPUT:**

list – two lists of labels

**EXAMPLES:**

The list of an permutation from iet:

sage: p1 = iet.Permutation('1 2 3', '3 1 2')
sage: p1.list()
[['1', '2', '3'], ['3', '1', '2']]
sage: p1.alphabet("abc")
sage: p1.list()
[['a', 'b', 'c'], ['c', 'a', 'b']]

Recovering the permutation from this list (and the alphabet):

sage: q1 = iet.Permutation(p1.list(),alphabet=p1.alphabet())
sage: p1 == q1
True

The list of a quadratic permutation:
```python
sage: p2 = iet.GeneralizedPermutation('g o o', 'd d g')
sage: p2.list()
[['g', 'o', 'o'], ['d', 'd', 'g']]
```

Recovering the permutation:

```python
sage: q2 = iet.GeneralizedPermutation(p2.list(), alphabet=p2.alphabet())
sage: p2 == q2
True
```

**rauzy_move_loser**(winner=None, side=None)

Returns the loser of a Rauzy move

**INPUT:**

- winner - either 'top' or 'bottom' ('t' or 'b' for short)
- side - either 'left' or 'right' ('l' or 'r' for short)

**OUTPUT:**

- a label

**EXAMPLES:**

```python
sage: p = iet.Permutation('a b c d', 'b d a c')
sage: p.rauzy_move_loser('top', 'right')
'c'
sage: p.rauzy_move_loser('bottom', 'right')
'd'
sage: p.rauzy_move_loser('top', 'left')
'b'
sage: p.rauzy_move_loser('bottom', 'left')
'a'
```

**rauzy_move_matrix**(winner=None, side='right')

Returns the Rauzy move matrix.

This matrix corresponds to the action of a Rauzy move on the vector of lengths. By convention (to get a positive matrix), the matrix is defined as the inverse transformation on the length vector.

**OUTPUT:**

matrix – a square matrix of positive integers

**EXAMPLES:**

```python
sage: p = iet.Permutation('a b', 'b a')
sage: p.rauzy_move_matrix('t')
[1 0]
[1 1]
sage: p.rauzy_move_matrix('b')
[1 1]
[0 1]
```

```python
sage: p = iet.Permutation('a b c d', 'b d a c')
sage: q = p.left_right_inverse()
sage: m0 = p.rauzy_move_matrix(winner='top', side='right')
sage: n0 = q.rauzy_move_matrix(winner='top', side='left')
sage: m0 == n0
```

(continues on next page)
True
sage: m1 = p.rauzy_move_matrix(winner='bottom', side='right')
sage: n1 = q.rauzy_move_matrix(winner='bottom', side='left')
sage: m1 == n1
True

rauzy_move_winner(winner=None, side=None)

Returns the winner of a Rauzy move.

INPUT:
- winner - either 'top' or 'bottom' ('t' or 'b' for short)
- side - either 'left' or 'right' ('l' or 'r' for short)

OUTPUT:
- a label

EXAMPLES:

sage: p = iet.Permutation('a b c d', 'b d a c')
sage: p.rauzy_move_winner('top', 'right')
'd'
sage: p.rauzy_move_winner('bottom', 'right')
'c'
sage: p.rauzy_move_winner('top', 'left')
'a'
sage: p.rauzy_move_winner('bottom', 'left')
'b'

sage: p = iet.GeneralizedPermutation('a b b c', 'd c a e d e')
sage: p.rauzy_move_winner('top', 'right')
'c'
sage: p.rauzy_move_winner('bottom', 'right')
'e'
sage: p.rauzy_move_winner('top', 'left')
'a'
sage: p.rauzy_move_winner('bottom', 'left')
'd'

class sage.dynamics.interval_exchanges.labelled.LabelledPermutationIET(intervals=None, alphabet=None)

Bases: sage.dynamics.interval_exchanges.labelled.LabelledPermutation, sage.dynamics.interval_exchanges.template.PermutationIET

Labelled permutation for iet

EXAMPLES:

Reducibility testing:

sage: p = iet.Permutation('a b c', 'c b a')
sage: p.is_irreducible()
True

sage: q = iet.Permutation('a b c d', 'b a d c')
Rauzy movability and Rauzy move:

```python
sage: p = iet.Permutation('a b c', 'c b a')
sage: p.has_rauzy_move('top')
True
sage: p.rauzy_move('bottom')
a c b
c b a
sage: p.has_rauzy_move('top')
True
sage: p.rauzy_move('top')
a b c
c a b
```

Rauzy diagram:

```python
sage: p = iet.Permutation('a b c', 'c b a')
sage: d = p.rauzy_diagram()
sage: p in d
True
```

`has_rauzy_move(winner=None, side=None)`

Returns True if you can perform a Rauzy move.

**INPUT:**

- `winner` - the winner interval ('top' or 'bottom')
- `side` - (default: 'right') the side ('left' or 'right')

**OUTPUT:**

bool – True if self has a Rauzy move

**EXAMPLES:**

```python
sage: p = iet.Permutation('a b','b a')
sage: p.has_rauzy_move()
True

sage: p = iet.Permutation('a b c','b a c')
sage: p.has_rauzy_move()
False
```

`is_identity()`

Returns True if self is the identity.

**OUTPUT:**

bool – True if self corresponds to the identity

**EXAMPLES:**

```python
sage: iet.Permutation("a b","a b").is_identity()
True
sage: iet.Permutation("a b","b a").is_identity()
False
```
rauzy_diagram(**args)
Returns the associated Rauzy diagram.
For more information try help(iet.RauzyDiagram).

OUTPUT:
Rauzy diagram – the Rauzy diagram of the permutation

EXAMPLES:

```python
sage: p = iet.Permutation('a b c', 'c b a')
sage: d = p.rauzy_diagram()
```

rauzy_move(winner=None, side=None, iteration=1)
Returns the Rauzy move.

INPUT:
• winner - the winner interval ('top' or 'bottom')
• side - (default: 'right') the side ('left' or 'right')

OUTPUT:
permutation – the Rauzy move of the permutation

EXAMPLES:

```python
sage: p = iet.Permutation('a b', 'b a')
sage: p.rauzy_move('t','right')
a b
b a
sage: p.rauzy_move('b','right')
a b
b a

sage: p = iet.Permutation('a b c', 'c b a')
sage: p.rauzy_move('t','right')
a b c
b c a
sage: p.rauzy_move('b','right')
b a c
b c a

sage: p = iet.Permutation('a b', 'b a')
sage: p.rauzy_move('t','left')
a b
b a
sage: p.rauzy_move('b','left')
a b
b a

sage: p = iet.Permutation('a b c', 'c b a')
sage: p.rauzy_move('t','left')
a b c
b c a
sage: p.rauzy_move('b','left')
b a c
b c a
```
rauzy_move_interval_substitution \((\text{winner}=\text{None}, \text{side}=\text{None})\)

Returns the interval substitution associated.

INPUT:

- \text{winner} - the winner interval (‘top’ or ‘bottom’)
- \text{side} - (default: ‘right’) the side (‘left’ or ‘right’)

OUTPUT:

WordMorphism – a substitution on the alphabet of the permutation

EXAMPLES:

```
sage: p = iet.Permutation('a b','b a')
sage: p.rauzy_move_interval_substitution('top','right')
WordMorphism: a->a, b->ba
sage: p.rauzy_move_interval_substitution('bottom','right')
WordMorphism: a->ab, b->b
sage: p.rauzy_move_interval_substitution('top','left')
WordMorphism: a->ba, b->b
sage: p.rauzy_move_interval_substitution('bottom','left')
WordMorphism: a->a, b->ab
```

rauzy_move_orbit_substitution \((\text{winner}=\text{None}, \text{side}=\text{None})\)

Return the action of the rauzy_move on the orbit.

INPUT:

- \text{i} - integer
- \text{winner} - the winner interval (‘top’ or ‘bottom’)
- \text{side} - (default: ‘right’) the side (‘right’ or ‘left’)

OUTPUT:

WordMorphism – a substitution on the alphabet of self

EXAMPLES:

```
sage: p = iet.Permutation('a b','b a')
sage: p.rauzy_move_orbit_substitution('top','right')
WordMorphism: a->ab, b->b
sage: p.rauzy_move_orbit_substitution('bottom','right')
WordMorphism: a->a, b->ab
sage: p.rauzy_move_orbit_substitution('top','left')
WordMorphism: a->a, b->ba
sage: p.rauzy_move_orbit_substitution('bottom','left')
WordMorphism: a->ba, b->b
```

reduced()

Returns the associated reduced abelian permutation.

OUTPUT:

a reduced permutation – the underlying reduced permutation

EXAMPLES:

```
sage: p = iet.Permutation("a b c d","d c a b")
sage: q = iet.Permutation("a b c d","d c a b",reduced=True)
```

(continues on next page)
class sage.dynamics.interval_exchanges.labelled.LabelledPermutationLI(intervals=None, alphabet=None):

Bases: sage.dynamics.interval_exchanges.labelled.LabelledPermutation, sage.dynamics.interval_exchanges.template.PermutationLI

Labelled quadratic (or generalized) permutation

EXAMPLES:

Reducibility testing:

```
sage: p = iet.GeneralizedPermutation('a b b', 'c c a')
sage: p.is_irreducible()
True
```

Reducibility testing with associated decomposition:

```
sage: p = iet.GeneralizedPermutation('a b c a', 'b d d c')
sage: p.is_irreducible()
False
sage: test, decomposition = p.is_irreducible(return_decomposition = True)
sage: test
False
sage: decomposition
([['a']], ['c', 'a'], [], ['c'])
```

Rauzy movability and Rauzy move:

```
sage: p = iet.GeneralizedPermutation('a a b b c c', 'd d')
sage: p.has_rauzy_move(0)
False
sage: p.has_rauzy_move(1)
True
sage: q = p.rauzy_move(1)
sage: q
a a b b c
  c d d
sage: q.has_rauzy_move(0)
True
sage: q.has_rauzy_move(1)
True
```

Rauzy diagrams:

```
sage: p = iet.GeneralizedPermutation('0 0 1 1', '2 2')
sage: r = p.rauzy_diagram()
sage: p in r
True
```

```
has_right_rauzy_move(winner)
    Test of Rauzy movability with a specified winner
```
A quadratic (or generalized) permutation is rauzy_movable type depending on the possible length of the last interval. It is dependent of the length equation.

**INPUT:**

- **winner** - ‘top’ (or ‘t’ or 0) or ‘bottom’ (or ‘b’ or 1)

**OUTPUT:**

bool – True if self has a Rauzy move

**EXAMPLES:**

```python
sage: p = iet.GeneralizedPermutation('a a','b b')

sage: p.has_right_rauzy_move('top')
False

sage: p.has_right_rauzy_move('bottom')
False

sage: p = iet.GeneralizedPermutation('a a b','b c c')

sage: p.has_right_rauzy_move('top')
True

sage: p.has_right_rauzy_move('bottom')
True

sage: p = iet.GeneralizedPermutation('a a b b','c c')

sage: p.has_right_rauzy_move('top')
False

sage: p.has_right_rauzy_move('bottom')
True
```

**left_rauzy_move**(winner)

Perform a Rauzy move on the left.

**INPUT:**

- **winner** - ‘top’ or ‘bottom’

**OUTPUT:**

permutation – the Rauzy move of self

**EXAMPLES:**

```python
sage: p = iet.GeneralizedPermutation('a a b','b c c')

sage: p.left_rauzy_move(0)
a a b b
b c c

sage: p.left_rauzy_move(1)
a a b
b c c

sage: p = iet.GeneralizedPermutation('a a b','c c a')

sage: p.left_rauzy_move(0)
(continues on next page)
```
rauzy_diagram(**kargs)

Returns the associated RauzyDiagram.

OUTPUT:
Rauzy diagram – the Rauzy diagram of the permutation

EXAMPLES:

\[
\begin{align*}
\text{sage: } & \text{p = iet.GeneralizedPermutation('a b c b', 'c d d a')} \\
\text{sage: } & \text{d = p.rauzy_diagram()} \\
\text{sage: } & \text{p in d} \\
\text{True}
\end{align*}
\]

For more information, try help(iet.RauzyDiagram)

reduced()

Returns the associated reduced quadratic permutations.

OUTPUT:
permutation – the underlying reduced permutation

EXAMPLES:

\[
\begin{align*}
\text{sage: } & \text{p = iet.GeneralizedPermutation('a a', 'b b c c')} \\
\text{sage: } & \text{q = p.reduced()} \\
\text{sage: } & \text{q} \\
\text{a a} \\
\text{b b c c} \\
\text{sage: } & \text{p.rauzy_move(0).reduced() == q.rauzy_move(0)} \\
\text{True}
\end{align*}
\]

right_rauzy_move(winner)

Perform a Rauzy move on the right (the standard one).

INPUT:
• winner - ‘top’ (or ‘t’ or 0) or ‘bottom’ (or ‘b’ or 1)

OUTPUT:
boolean – True if self has a Rauzy move

EXAMPLES:

\[
\begin{align*}
\text{sage: } & \text{p = iet.GeneralizedPermutation('a a b', 'b c c')} \\
\text{sage: } & \text{p.right_rauzy_move(0)} \\
\text{a a b} \\
\text{b c c} \\
\text{sage: } & \text{p.right_rauzy_move(1)} \\
\text{a a} \\
\text{b b c c}
\end{align*}
\]
sage: p = iet.GeneralizedPermutation('a b b','c c a')
sage: p.right_rauzy_move(0)
a a b b
c c
sage: p.right_rauzy_move(1)
a b b
c c a

sage.dynamics.interval_exchanges.labelled.LabelledPermutationsIET_iterator (nintervals=None, irreducible=True, alphabet=None)

Returns an iterator over labelled permutations.

INPUT:
- nintervals - integer or None
- irreducible - boolean (default: True)
- alphabet - something that should be converted to an alphabet of at least nintervals letters

OUTPUT:
iterator – an iterator over permutations

class sage.dynamics.interval_exchanges.labelled.LabelledRauzyDiagram (p, right_induction=True, left_induction=False, left_right_inversion=False, top_bottom_inversion=False, symmetric=False)

Bases: sage.dynamics.interval_exchanges.template.RauzyDiagram

Template for Rauzy diagrams of labelled permutations.

Warning: DO NOT USE

class Path (parent, *data)
Bases: sage.dynamics.interval_exchanges.template.RauzyDiagram.Path

Path in Labelled Rauzy diagram.

dual_substitution ()

Returns the substitution of intervals obtained.

OUTPUT:
WordMorphism – the word morphism corresponding to the interval

EXAMPLES:

sage: p = iet.Permutation('a b','b a')
sage: r = p.rauzy_diagram()
interval_substitution()

Returns the substitution of intervals obtained.

OUTPUT:

WordMorphism – the word morphism corresponding to the interval

EXAMPLES:

sage: p = iet.Permutation('a b','b a')
sage: r = p.rauzy_diagram()
sage: p0 = r.path(p,0)
sage: s0 = p0.interval_substitution()
sage: s0
WordMorphism: a->a, b->ba
sage: p1 = r.path(p,1)
sage: s1 = p1.interval_substitution()
sage: s1
WordMorphism: a->ab, b->b
sage: (p0 + p1).interval_substitution() == s1 * s0
True
sage: (p1 + p0).interval_substitution() == s0 * s1
True

is_full()

Tests the fullness.

A path is full if all intervals win at least one time.

OUTPUT:

boolean – True if the path is full and False else

EXAMPLES:

sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: g0 = r.path(p,'t','b','t')
sage: g1 = r.path(p,'b','t','b')
sage: g0.is_full()
False
sage: g1.is_full()
False
sage: (g0 + g1).is_full()
True
matrix()

Returns the matrix associated to a path.

The matrix associated to a Rauzy induction, is the linear application that allows to recover the lengths of `self` from the lengths of the induced.

OUTPUT:

matrix – a square matrix of integers

EXAMPLES:

```python
sage: p = iet.Permutation('a1 a2','a2 a1')
sage: d = p.rauzy_diagram()
sage: g = d.path(p, 'top')
sage: g.matrix()
[1 0]
[1 1]
sage: g = d.path(p, 'bottom')
sage: g.matrix()
[1 1]
[0 1]
```

```python
sage: p = iet.Permutation('a b c','c b a')
sage: d = p.rauzy_diagram()
sage: g = d.path(p)  # g.matrix() == identity_matrix(3) is True
sage: g.matrix()
[1 0 0]
[0 1 0]
[1 0 1]
sage: g = d.path(p, 'top')
sage: g.matrix()
[1 0 0]
[0 1 0]
[1 0 1]
sage: g = d.path(p, 'bottom')
sage: g.matrix()
[1 0 1]
[0 1 0]
[0 0 1]
```

orbit_substitution()

Return the substitution on the orbit of the left extremity.

OUTPUT:

WordMorphism – the word morphism corresponding to the orbit

EXAMPLES:

```python
sage: p = iet.Permutation('a b','b a')
sage: d = p.rauzy_diagram()
sage: g0 = d.path(p, 'top')
sage: s0 = g0.orbit_substitution()
sage: s0
WordMorphism: a->ab, b->b
sage: gl = d.path(p, 'bottom')
```

(continues on next page)
substitution()
Return the substitution on the orbit of the left extremity.

OUTPUT:
WordMorphism – the word morphism corresponding to the orbit

EXAMPLES:

```python
sage: p = iet.Permutation('a b','b a')
sage: d = p.rauzy_diagram()
sage: g0 = d.path(p,'top')
sage: s0 = g0.orbit_substitution()
sage: s0
WordMorphism: a->ab, b->b
sage: g1 = d.path(p,'bottom')
sage: s1 = g1.orbit_substitution()
sage: s1
WordMorphism: a->a, b->ab
sage: (g0 + g1).orbit_substitution() == s0 * s1
True
sage: (g1 + g0).orbit_substitution() == s1 * s0
True
```

edge_to_interval_substitution(p=None, edge_type=None)
Returns the interval substitution associated to an edge

OUTPUT:
WordMorphism – the WordMorphism corresponding to the edge

EXAMPLES:

```python
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: r.edge_to_interval_substitution(None,None)
WordMorphism: a->a, b->b, c->c
sage: r.edge_to_interval_substitution(p,0)
WordMorphism: a->a, b->b, c->ca
sage: r.edge_to_interval_substitution(p,1)
WordMorphism: a->ac, b->b, c->c
```

edge_to_orbit_substitution(p=None, edge_type=None)
Returns the interval substitution associated to an edge

OUTPUT:
WordMorphism – the word morphism corresponding to the edge

EXAMPLES:
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: r.edge_to_orbit_substitution(None,None)
WordMorphism: a->a, b->b, c->c

sage: r.edge_to_orbit_substitution(p,0)
WordMorphism: a->ac, b->b, c->c

sage: r.edge_to_orbit_substitution(p,1)
WordMorphism: a->a, b->b, c->ac

**full_loop_iterator** *(start=None, max_length=1)*

Returns an iterator over all full path starting at start.

**INPUT:**

- **start** - the start point
- **max_length** - a limit on the length of the paths

**OUTPUT:**

iterator – iterator over full loops

**EXAMPLES:**

sage: p = iet.Permutation('a b','b a')
sage: d = p.rauzy_diagram()
sage: for g in d.full_loop_iterator(p,2):
    ....: print(g.matrix())
    ....: print("*****")
[1 1]
[1 2]
*****
[2 1]
[1 1]
*****

**full_nloop_iterator** *(start=None, length=1)*

Returns an iterator over all full loops of given length.

**INPUT:**

- **start** - the initial permutation
- **length** - the length to consider

**OUTPUT:**

iterator – an iterator over the full loops of given length

**EXAMPLES:**

sage: p = iet.Permutation('a b','b a')
sage: d = p.rauzy_diagram()
sage: for g in d.full_nloop_iterator(p,2):
    ....: print(g.matrix())
    ....: print("*****")
[1 1]
[1 2]
*****
[2 1]
[1 1]
*****
4.3 Reduced permutations

**Warning:** This module is deprecated. You are advised to install and use the surface_dynamics package instead available at https://pypi.python.org/pypi/surface_dynamics/

A reduced (generalized) permutation is better suited to study strata of Abelian (or quadratic) holomorphic forms on Riemann surfaces. The Rauzy diagram is an invariant of such a component. Corentin Boissy proved the identification of Rauzy diagrams with connected components of stratas. But the geometry of the diagram and the relation with the strata is not yet totally understood.

**AUTHORS:**

- Vincent Delecroix (2000-09-29): initial version

```python
class sage.dynamics.interval_exchanges.reduced.FlippedReducedPermutation:
    Bases: sage.dynamics.interval_exchanges.reduced.ReducedPermutation

Flipped Reduced Permutation.

_WARNING: Internal class! Do not use directly!
```

**INPUT:**

- **intervals** - a list of two lists
- **flips** - the flipped letters
- **alphabet** - an alphabet

```python
def right_rauzy_move(self, winner):
    pass  # Perform a Rauzy move on the right.
```

**EXAMPLES:**

```python
sage: p = iet.Permutation('a b c', 'c b a', reduced=True, flips='c')
sage: p.right_rauzy_move('top')
-a b -c
-a -c b
```

```python
class sage.dynamics.interval_exchanges.reduced.FlippedReducedPermutationIET:
    Bases: sage.dynamics.interval_exchanges.reduced.FlippedReducedPermutation,
           sage.dynamics.interval_exchanges.template.FlippedPermutationIET,
           sage.dynamics.interval_exchanges.reduced.ReducedPermutationIET

Flipped Reduced Permutation from iet

EXAMPLES
```
```
sage: p = iet.Permutation('a b c', 'c b a', flips=['a'], reduced=True)
sage: p.rauzy_move(1)
-a -b c
-a  c -b
```

**list** *(flips=False)*

Returns a list representation of self.

**INPUT:**

- **flips** - boolean (default: False) if True the output contains 2-uple of (label, flip)

**EXAMPLES:**

```
sage: p = iet.Permutation('a b','b a',reduced=True,flips='b')
sage: p.list(flips=True)
[['a', 1], ['b', -1]]
sage: p.list(flips=False)
['a', 'b']
sage: p.alphabet([0,1])
sage: p.list(flips=True)
[(0, 1), (1, -1)]
sage: p.list(flips=False)
[0, 1]
```

One can recover the initial permutation from this list:

```
sage: p = iet.Permutation('a b','b a',reduced=True,flips='a')
sage: iet.Permutation(p.list(), flips=p.flips(), reduced=True) == p
True
```

**rauzy_diagram** (**kargs**)

Returns the associated Rauzy diagram.

**EXAMPLES:**

```
sage: p = iet.Permutation('a b','b a',reduced=True,flips='a')
sage: r = p.rauzy_diagram()
sage: p in r
True
```

```
sage: p = iet.Permutation('a b','b a',reduced=True,flips='a')
sage: r = p.rauzy_diagram()
sage: p in r
True
```

```
sage: p = iet.Permutation('a a b', 'b c c', reduced=True, flips='a')
sage: p.list(flips=False)
['a', 'a', 'b']
```

4.3. Reduced permutations

```
sage: p = iet.GeneralizedPermutation('a a b', 'c b a', reduced=True, flips=['a'])
sage: p.list(flips=False)
-a -b c
-a  c -b
```

```
sage: p = iet.GeneralizedPermutation('a a b', 'b c c', reduced=True, flips=['a'])
sage: p.list(flips=False)
-a -b c
-a  c -b
```

**list** *(flips=False)*

Returns a list representation of self.
INPUT:

- `flips` - boolean (default: False) return the list with flips

EXAMPLES:

```python
sage: p = iet.GeneralizedPermutation('a a','b b',reduced=True,flips='a')
sage: p.list(flips=True)
[['a', -1], ['a', -1]]
sage: p.list(flips=False)
[['a', 'a'], ['b', 'b']]
```

one can rebuild the permutation from the list:

```python
sage: p = iet.GeneralizedPermutation('a a b','b c c',flips='a',reduced=True)
sage: iet.GeneralizedPermutation(p.list(),flips=p.flips(),reduced=True) == p
True
```

```python
def rauzy_diagram(**kargs):
    Returns the associated Rauzy diagram.
    For more explanation and a list of arguments try help(iet.RauzyDiagram)
```

```python
sage: p = iet.GeneralizedPermutation('a a b','c c b',reduced=True)
sage: r = p.rauzy_diagram()
sage: p in r
True
```

```
class sage.dynamics.interval_exchanges.reduced.FlippedReducedRauzyDiagram(p,
    right_induction=True,
    left_induction=False,
    left_right_inversion=False,
    top_bottom_inversion=False,
    symmetric=False, **kargs):
    Bases: sage.dynamics.interval_exchanges.template.FlippedRauzyDiagram, sage.dynamics.interval_exchanges.reduced.ReducedRauzyDiagram
    Rauzy diagram of flipped reduced permutations.
```

```
class sage.dynamics.interval_exchanges.reduced.ReducedPermutation(intervals=None,
    alpha_beta=None, **kargs):
    Bases: sage.structure.sage_object.SageObject
    Template for reduced objects.

    Warning: Internal class! Do not use directly!
```

INPUT:

### Chapter 4. Interval exchange transformations and linear involutions
• intervals - a list of two list of labels
• alphabet - (default: None) any object that can be used to initialize an Alphabet or None. In this latter case, the letter of the intervals are used to generate one.

`erase_letter(letter)`
Erases a letter.

**INPUT:**

• letter - a letter which is a label of an interval of self

**EXAMPLES:**

```
sage: p = iet.Permutation('a b c','c b a')
sage: p.erase_letter('a')
b c
c b
```

```
sage: p = iet.GeneralizedPermutation('a b b','c c a')
sage: p.erase_letter('a')
b b
c c
```

`left_rauzy_move(winner)`
Performs a Rauzy move on the left.

**EXAMPLES:**

```
sage: p = iet.Permutation('a b c','c b a')
sage: p.left_rauzy_move(0)
a b c
b c a
sage: p.right_rauzy_move(1)
a b c
b c a
```

```
sage: p = iet.GeneralizedPermutation('a a','b b c c')
sage: p.left_rauzy_move(0)
a a b
b c c
```

`length(interval=None)`
Returns the 2-uple of lengths.

p.length() is identical to (p.length_top(), p.length_bottom()) If an interval is specified, it returns the length of the specified interval.

**INPUT:**

• interval - None, ‘top’ (or ‘t’ or 0) or ‘bottom’ (or ‘b’ or 1)

**OUTPUT:**
integer or 2-uple of integers – the corresponding lengths

**EXAMPLES:**

```
sage: p = iet.Permutation('a b c','c b a')
sage: p.length()
(3, 3)
```
(continues on next page)
sage: p = iet.GeneralizedPermutation('a a b','c d c b d')
sage: p.length()
(3, 5)

length_bottom()  
Returns the number of intervals in the bottom segment.

OUTPUT:  
integer – the length of the bottom segment

EXAMPLES:

sage: p = iet.Permutation('a b c','c b a')
sage: p.length_bottom()
3
sage: p = iet.GeneralizedPermutation('a a b','c d c b d')
sage: p.length_bottom()
5

length_top()  
Returns the number of intervals in the top segment.

OUTPUT:  
integer – the length of the top segment

EXAMPLES:

sage: p = iet.Permutation('a b c','c b a')
sage: p.length_top()
3
sage: p = iet.GeneralizedPermutation('a a b','c d c b d')
sage: p.length_top()
3
sage: p = iet.GeneralizedPermutation('a b c b d c d', 'e a e')
sage: p.length_top()
7

right_rauzy_move(winner)  
Performs a Rauzy move on the right.

EXAMPLES:

sage: p = iet.Permutation('a b c','c b a',reduced=True)
sage: p.right_rauzy_move(0)
a b c
   c a b
sage: p.right_rauzy_move(1)
a b c
   b c a

sage: p = iet.GeneralizedPermutation('a a','b b c c',reduced=True)
sage: p.right_rauzy_move(0)
a b b
   c c a
class sage.dynamics.interval_exchanges.reduced.ReducedPermutationIET(intervals=None, alpha=None, beta=None):

Bases: sage.dynamics.interval_exchanges.reduced.ReducedPermutation, sage.dynamics.interval_exchanges.template.PermutationIET

Reduced permutation from iet
Permutation from iet without numerotation of intervals. For initialization, you should use GeneralizedPermutation which is the class factory for all permutation types.

EXAMPLES:

Equality testing (no equality of letters but just of ordering):

```python
sage: p = iet.Permutation('a b c', 'c b a', reduced = True)
sage: q = iet.Permutation('p q r', 'r q p', reduced = True)
sage: p == q
True
```

Reducibility testing:

```python
sage: p = iet.Permutation('a b c', 'c b a', reduced = True)
sage: p.is_irreducible()
True

sage: q = iet.Permutation('a b c d', 'b a d c', reduced = True)
sage: q.is_irreducible()
False
```

Rauzy movability and Rauzy move:

```python
sage: p = iet.Permutation('a b c', 'c b a', reduced = True)
sage: p.has_rauzy_move(1)
True

sage: p.rauzy_move(1)
a b c
b c a
```

Rauzy diagrams:

```python
sage: p = iet.Permutation('a b c d', 'd a b c')
sage: p_red = iet.Permutation('a b c d', 'd a b c', reduced = True)
sage: d = p.rauzy_diagram()
sage: d_red = p_red.rauzy_diagram()
sage: p.rauzy_move(0) in d
True

sage: d.cardinality(), d_red.cardinality()
(12, 6)
```

has_rauzy_move(winner, side='right')
Tests if the permutation is rauzy_movable on the left.

EXAMPLES:

```python
sage: p = iet.Permutation('a b c', 'a c b', reduced=True)
sage: p.has_rauzy_move(0, 'right')
True

sage: p.has_rauzy_move(0, 'left')
```

(continues on next page)
False
sage: p.has_rauzy_move(1,'right')
True
sage: p.has_rauzy_move(1,'left')
False

sage: p = iet.Permutation('a b c d','c a b d',reduced=True)
sage: p.has_rauzy_move(0,'right')
False
sage: p.has_rauzy_move(0,'left')
True
sage: p.has_rauzy_move(1,'right')
False
sage: p.has_rauzy_move(1,'left')
True

is_identity()
Returns True if self is the identity.

EXAMPLES:

sage: iet.Permutation("a b","a b",reduced=True).is_identity()
True
sage: iet.Permutation("a b","b a",reduced=True).is_identity()
False

list()
Returns a list of two list that represents the permutation.

EXAMPLES:

sage: p = iet.GeneralizedPermutation('a b','b a',reduced=True)
sage: p.list() == [p[0], p[1]]
True
sage: p.list() == [['a', 'b'], ['b', 'a']]
True

sage: p = iet.GeneralizedPermutation('a b c', 'b c a',reduced=True)
sage: iet.GeneralizedPermutation(p.list(),reduced=True) == p
True

rauzy_diagram(**kargs)
Returns the associated Rauzy diagram.

OUTPUT:
A Rauzy diagram

EXAMPLES:

sage: p = iet.Permutation('a b c d', 'd a b c',reduced=True)
sage: d = p.rauzy_diagram()
sage: p.rauzy_move(0) in d
True
sage: p.rauzy_move(1) in d
True

For more information, try help RauzyDiagram
\textbf{rauzy_move_relabel} \texttt{(winner, side='right')}

Returns the relabelization obtained from this move.

\textbf{EXAMPLES:}

```
sage: p = iet.Permutation('a b c d', 'd c b a')
sage: q = p.reduced()
sage: p_t = p.rauzy_move('t')
sage: q_t = q.rauzy_move('t')
sage: s_t = q.rauzy_move_relabel('t')
sage: s_t
WordMorphism: a->a, b->b, c->c, d->d
sage: list(map(s_t, p_t[0])) == list(map(Word, q_t[0]))
True
sage: list(map(s_t, p_t[1])) == list(map(Word, q_t[1]))
True
sage: p_b = p.rauzy_move('b')
sage: q_b = q.rauzy_move('b')
sage: s_b = q.rauzy_move_relabel('b')
sage: s_b
WordMorphism: a->a, b->d, c->b, d->c
sage: list(map(s_b, q_b[0])) == list(map(Word, p_b[0]))
True
sage: list(map(s_b, q_b[1])) == list(map(Word, p_b[1]))
True
```

\textbf{class} \texttt{sage.dynamics.interval_exchanges.reduced.ReducedPermutationLI} \texttt{(intervals=None, alpha=None, beta=None)}

\texttt{Bases: sage.dynamics.interval_exchanges.reduced.ReducedPermutation, sage.dynamics.interval_exchanges.template.PermutationLI}

Reduced quadratic (or generalized) permutation.

\textbf{EXAMPLES:}

Reducibility testing:

```
sage: p = iet.GeneralizedPermutation('a b b', 'c c a', reduced = True)
sage: p.is_irreducible()
True
```

```
sage: p = iet.GeneralizedPermutation('a b c a', 'b d d c', reduced = True)
sage: p.is_irreducible()
False
sage: test, decomposition = p.is_irreducible(return_decomposition = True)
sage: test
False
sage: decomposition
(['a'], ['c', 'a'], [], ['c'])
```

Rauzy movability and Rauzy move:

```
sage: p = iet.GeneralizedPermutation('a b b', 'c c a', reduced = True)
sage: p.has_rauzy_move(0)
True
sage: p.rauzy_move(0)
a a b b
c c
```

(continues on next page)
Rauzy diagrams:

```python
sage: p_red = iet.GeneralizedPermutation('a b b', 'c c a', reduced = True)
sage: d_red = p_red.rauzy_diagram()
sage: d_red.cardinality()
4
```

list()  
The permutations as a list of two lists.

EXAMPLES:

```python
sage: p = iet.GeneralizedPermutation('a b b', 'c c a', reduced = True)
sage: list(p)
[['a', 'b', 'b'], ['c', 'c', 'a']]
```

rauzy_diagram(**kargs)  
Returns the associated Rauzy diagram.

The Rauzy diagram of a permutation corresponds to all permutations that we could obtain from this one by Rauzy move. The set obtained is a labelled Graph. The label of vertices being 0 or 1 depending on the type.

OUTPUT:
Rauzy diagram – the graph of permutations obtained by rauzy induction

EXAMPLES:

```python
sage: p = iet.Permutation('a b c d', 'd a b c')
sage: d = p.rauzy_diagram()
```

sage.dynamics.interval_exchanges.reduced.ReducedPermutationsIET_iterator (nintervals=None, irreducible=True, alphabet=None)

Returns an iterator over reduced permutations

INPUT:
- nintervals - integer or None
- irreducible - boolean
- alphabet - something that should be converted to an alphabet of at least nintervals letters
class sage.dynamics.interval_exchanges.reduced.ReducedRauzyDiagram(p,  
right_induction=True,  
left_induction=False,  
left_right_inversion=False,  
top_bottom_inversion=False,  
symmetric=False)

Bases: sage.dynamics.interval_exchanges.template.RauzyDiagram

Rauzy diagram of reduced permutations

sage.dynamics.interval_exchanges.reduced.alphabetized_atwin(twin, alphabet)
Alphabetization of a twin of iet.

```
sage: twin = [[0,1],[0,1]]
sage: alphabet = Alphabet("ab")
sage: alphabetized_atwin(twin, alphabet)
[['a', 'b'], ['a', 'b']]
```

```
sage: twin = [[1,0],[1,0]]
sage: alphabet = Alphabet([0,1])
sage: alphabetized_atwin(twin, alphabet)
[[0, 1], [1, 0]]
```

```
sage: twin = [[1,2,3,0],[3,0,1,2]]
sage: alphabet = Alphabet("abcd")
sage: alphabetized_atwin(twin,alphabet)
[['a', 'b', 'c', 'd'], ['d', 'a', 'b', 'c']]
```

sage.dynamics.interval_exchanges.reduced.alphabetized_qtwin(twin, alphabet)
Alphabetization of a qtwin.

```
sage: twin = [[[1,0),(1,1)],[(0,0),(0,1)]]
sage: alphabet = Alphabet("ab")
sage: alphabetized_qtwin(twin,alphabet)
[['a', 'b'], ['a', 'b']]  
```

```
sage: twin = [[[1,1),(1,0)],[(0,1),(0,0)]]
sage: alphabet=Alphabet("BA")
sage: alphabetized_qtwin(twin,alphabet)
[['B', 'A'], ['A', 'B']]
```

```
sage: twin = [[[0,1),(0,0)],[(1,1),(1,0)]]
sage: alphabet=Alphabet("ab")
sage: alphabetized_qtwin(twin,alphabet)
[['a', 'a'], ['b', 'b']]
```

```
sage: twin = [[[0,2),(1,1),(0,0)],[(1,2),(0,1),(1,0)]]
sage: alphabet=Alphabet("abc")
sage: alphabetized_qtwin(twin,alphabet)
[['a', 'b', 'a'], ['c', 'b', 'c']]  
```

sage.dynamics.interval_exchanges.reduced.labelize_flip(couple)
Returns a string from a 2-uple couple of the form (name, flip).

4.3. Reduced permutations 81
4.4 Permutations template

Warning: This module is deprecated. You are advised to install and use the surface_dynamics package instead available at https://pypi.python.org/pypi/surface_dynamics/

This file defines high level operations on permutations (alphabet, the different rauzy induction, ...) shared by reduced and labeled permutations.

AUTHORS:
• Vincent Delecroix (2008-12-20): initial version

Todo:
• construct as options different string representations for a permutation
  – the two intervals: str
  – the two intervals on one line: str_one_line
  – the separatrix diagram: str_separatrix_diagram
  – twin[0] and twin[1] for reduced permutation
  – nothing (useful for Rauzy diagram)

class sage.dynamics.interval_exchanges.template.FlippedPermutation
  Bases: sage.dynamics.interval_exchanges.template.Permutation
  Template for flipped generalized permutations.

  Warning: Internal class! Do not use directly!

  AUTHORS:
  • Vincent Delecroix (2008-12-20): initial version

  str(sep='\n')
  String representation.

class sage.dynamics.interval_exchanges.template.FlippedPermutationIET
  Bases: sage.dynamics.interval_exchanges.template.FlippedPermutation, sage.dynamics.interval_exchanges.template.PermutationIET
  Template for flipped Abelian permutations.

  Warning: Internal class! Do not use directly!

  AUTHORS:
  • Vincent Delecroix (2008-12-20): initial version

  flips()
  Returns the list of flips.

  EXAMPLES:
```python
sage: p = iet.Permutation('a b c', 'c b a', flips='ac')
sage: p.flips()
['a', 'c']
```

class sage.dynamics.interval_exchanges.template.FlippedPermutationLI

Bases: sage.dynamics.interval_exchanges.template.FlippedPermutation, sage.dynamics.interval_exchanges.template.PermutationLI

Template for flipped quadratic permutations.

**Warning:** Internal class! Do not use directly!

**AUTHORS:**

• Vincent Delecroix (2008-12-20): initial version

**flips()**

Returns the list of flipped intervals.

**EXAMPLES:**

```python
sage: p = iet.GeneralizedPermutation('a a', 'b b', flips='a')
sage: p.flips()
['a']
sage: p = iet.GeneralizedPermutation('a a', 'b b', flips='b', reduced=True)
sage: p.flips()
['b']
```

class sage.dynamics.interval_exchanges.template.FlippedRauzyDiagram

Bases: sage.dynamics.interval_exchanges.template.RauzyDiagram

Template for flipped Rauzy diagrams.

**Warning:** Internal class! Do not use directly!

**AUTHORS:**

• Vincent Delecroix (2009-09-29): initial version

**complete** *(p, reducible=False)*

Completion of the Rauzy diagram

Add all successors of p for defined operations in edge_types. Could be used for generating non (strongly) connected Rauzy diagrams. Sometimes, for flipped permutations, the maximal connected graph in all permutations is not strongly connected. Finding such components needs to call most than once the .complete() method.

**INPUT:**

• p - a permutation
• reducible - put or not reducible permutations

EXAMPLES:

```python
sage: p = iet.Permutation('a b c','c b a',flips='a')
sage: d = p.rauzy_diagram()
sage: d
Rauzy diagram with 3 permutations
sage: p = iet.Permutation('a b c','c b a',flips='b')
sage: d.complete(p)
sage: d
Rauzy diagram with 8 permutations
```

```python
sage: p = iet.Permutation('a b c','c b a',flips='a')
sage: d.complete(p)
sage: d
Rauzy diagram with 8 permutations
```

class sage.dynamics.interval_exchanges.template.Permutation

Bases: sage.structure.sage_object.SageObject

Template for all permutations.

**Warning:** Internal class! Do not use directly!

This class implement generic algorithm (stratum, connected component, …) and unifies all its children.

**alphabet**(data=None)

Manages the alphabet of self.

If there is no argument, the method returns the alphabet used. If the argument could be converted to an alphabet, this alphabet will be used.

**INPUT:**

• data - None or something that could be converted to an alphabet

**OUTPUT:**

– either None or the current alphabet

**EXAMPLES:**

```python
sage: p = iet.Permutation('a b','a b')
sage: p.alphabet([0,1])
sage: p.alphabet() == Alphabet([0,1])
True
sage: p
0 1
0 1
sage: p.alphabet("cd")
sage: p.alphabet() == Alphabet(['c','d'])
True
sage: p
c d
c d
```

**has_rauzy_move**(winner='top', side=None)

Tests the legality of a Rauzy move.

**INPUT:**
• winner - ‘top’ or ‘bottom’ corresponding to the interval
• side - ‘left’ or ‘right’ (default)

OUTPUT:
– a boolean

EXAMPLES:

```sage
sage: p = iet.Permutation('a b','a b')
sage: p.has_rauzy_move('top','right')
False
sage: p.has_rauzy_move('bottom','right')
False
sage: p.has_rauzy_move('top','left')
False
sage: p.has_rauzy_move('bottom','left')
False

sage: p = iet.Permutation('a b c','b a c')
sage: p.has_rauzy_move('top','right')
False
sage: p.has_rauzy_move('bottom','right')
False
sage: p.has_rauzy_move('top','left')
True
sage: p.has_rauzy_move('bottom','left')
True
```

horizontal_inverse()
Returns the top-bottom inverse.
You can use also use the shorter .tb_inverse().

OUTPUT:
– a permutation

EXAMPLES:

```sage
sage: p = iet.Permutation('a b','b a')
sage: p.top_bottom_inverse()
b a
 a b
sage: p = iet.Permutation('a b','b a',reduced=True)
sage: p.top_bottom_inverse() == p
True
```
left_right_inverse()  
Returns the left-right inverse.

You can also use the shorter .lr_inverse()  

OUTPUT:  
– a permutation

EXAMPLES:

```python
sage: p = iet.Permutation('a b c','c a b')
sage: p.left_right_inverse()
c b a  
b a c
```

```python
sage: p = iet.Permutation('a b c d','c d a b')
sage: p.left_right_inverse()
d c b a  
b a d c
```

```python
sage: p = iet.GeneralizedPermutation('a a','b b c c')
sage: p.left_right_inverse()
a a  
c c b b
```

```python
sage: p = iet.Permutation('a b c','c b a',reduced=True)
sage: p.left_right_inverse() == p
True
```

```python
sage: p = iet.Permutation('a b c','c a b',reduced=True)
sage: q = p.left_right_inverse()
sage: q == p
False
```

```python
sage: p = iet.GeneralizedPermutation('a a','b b c c',reduced=True)
sage: p.left_right_inverse() == p
True
```

```python
sage: p = iet.GeneralizedPermutation('a b b','c c a',reduced=True)
sage: q = p.left_right_inverse()
sage: q == p
False
```

letters()  
Returns the list of letters of the alphabet used for representation.

The letters used are not necessarily the whole alphabet (for example if the alphabet is infinite).

OUTPUT:
– a list of labels

EXAMPLES:

```python
sage: p = iet.Permutation([1,2],[2,1])
sage: p.alphabet(Alphabet(name="NN"))
sage: p
0 1
1 0
sage: p.letters()
[0, 1]
```

```python
lr_inverse()

Returns the left-right inverse.

You can also use the shorter `lr_inverse()`

OUTPUT:
– a permutation

EXAMPLES:

```python
sage: p = iet.Permutation('a b c','c a b')
sage: p.left_right_inverse()
c b a
b a c
sage: p = iet.Permutation('a b c d','c d a b')
sage: p.left_right_inverse()
d c b a
b a d c
sage: p = iet.GeneralizedPermutation('a a','b b c c')
sage: p.left_right_inverse()
a a
c c b b
```

```python
sage: p = iet.Permutation('a b c','c b a',reduced=True)
sage: p.left_right_inverse() == p
True
sage: p = iet.Permutation('a b c','c a b',reduced=True)
sage: q = p.left_right_inverse()
sage: q == p
False
sage: q
a b c
b c a
sage: p = iet.GeneralizedPermutation('a a','b b c c',reduced=True)
sage: p.left_right_inverse() == p
True
sage: p = iet.GeneralizedPermutation('a b b','c c a',reduced=True)
sage: q = p.left_right_inverse()
sage: q == p
False
sage: q
a a b
b c c
```

4.4. Permutations template
rauzy_move \( (\text{winner, side='right', iteration=1}) \)
Returns the permutation after a Rauzy move.

**INPUT:**
- **winner** - 'top' or 'bottom' interval
- **side** - 'right' or 'left' (default: 'right') corresponding to the side on which the Rauzy move must be performed.
- **iteration** - a non negative integer

**OUTPUT:**
- a permutation

```sage
p = iet.Permutation('a b c','c b a')
p.rauzy_move(winner=0, side='right')
a b c
c a b
p.rauzy_move(winner=1, side='right')
a c b
b a c
p.rauzy_move(winner=0, side='left')
 a b c
b c a
p.rauzy_move(winner=1, side='left')
 b a c
 c b a
```

str \( (\text{sep='n'}) \)
A string representation of the generalized permutation.

**INPUT:**
- **sep** - (default: 'n') a separator for the two intervals

**OUTPUT:**
string – the string that represents the permutation

**EXAMPLES:**

For permutations of iet:

```sage
p = iet.Permutation('a b c','c b a')
p.str()
'a b c
 c b a'
p.str(sep=' | ')
'a b c | c b a'
```

..the permutation can be rebuilt from the standard string:

```sage
p == iet.Permutation(p.str())
True
```

For permutations of li:

```sage
p = iet.GeneralizedPermutation('a b b','c c a')
doctest:warning
...
DeprecationWarning: GeneralizedPermutation is deprecated and will be removed from Sage.
(continues on next page)```
You are advised to install the surface_dynamics package via:
```
sage -pip install surface_dynamics
```
If you do not have write access to the Sage installation you can
alternatively do
```
sage -pip install surface_dynamics --user
```
The package surface_dynamics subsumes all flat surface related
computation that are currently available in Sage. See more
information at
http://www.labri.fr/perso/vdelecro/surface-dynamics/latest/
See http://trac.sagemath.org/20695 for details.
```
sage: p.str()
'a b b \n c c a'
sage: p.str(sep=' | ')
'a b b | c c a'
```

.. the generalized permutation can be rebuilt from the standard string:
```
sage: p == iet.GeneralizedPermutation(p.str())
True
```

**symmetric()**

Returns the symmetric permutation.

The symmetric permutation is the composition of the top-bottom inversion and the left-right inversion
(which are geometrically orientation reversing).

**OUTPUT:**

– a permutation

**EXAMPLES:**
```
sage: p = iet.Permutation("a b c","c b a")
sage: p.symmetric()
a b c
c b a
sage: q = iet.Permutation("a b c d","b d a c")
sage: q.symmetric()
c a d b
d c b a
```
```
sage: p = iet.Permutation('a b c d','c a d b')
sage: q = p.symmetric()
sage: q1 = p.tb_inverse().lr_inverse()
sage: q2 = p.lr_inverse().tb_inverse()
sage: q == q1
True
sage: q == q2
True
sage: p = iet.GeneralizedPermutation('a a b','b c c',reduced=True)
sage: q = p.symmetric()
sage: q1 = p.tb_inverse().lr_inverse()
sage: q2 = p.lr_inverse().tb_inverse()
sage: q == q1
True
```

(continues on next page)
sage: q == q2
True

sage: p = iet.GeneralizedPermutation('a a b', 'b c c', reduced=True, flips='a')

sage: q = p.symmetric()

sage: q1 = p.tb_inverse().lr_inverse()

sage: q2 = p.lr_inverse().tb_inverse()

sage: q == q1
True

sage: q == q2
True

\texttt{tb\_inverse()} \\
Returns the top-bottom inverse.

You can use also use the shorter \texttt{.tb\_inverse()}.

\textbf{OUTPUT:}

– a permutation

\textbf{EXAMPLES:}

\begin{verbatim}
sage: p = iet.Permutation('a b', 'b a')
sage: p.top_bottom_inverse()
b a
a b

sage: p = iet.Permutation('a b', 'b a', reduced=True)
sage: p.top_bottom_inverse() == p
True

sage: p = iet.Permutation('a b c d', 'c d a b')
sage: p.top_bottom_inverse()
c d a b
a b c d
\end{verbatim}

\texttt{top\_bottom\_inverse()} \\
Returns the top-bottom inverse.

You can use also use the shorter \texttt{.tb\_inverse()}.

\textbf{OUTPUT:}

– a permutation

\textbf{EXAMPLES:}

\begin{verbatim}
sage: p = iet.Permutation('a b', 'b a')
sage: p.top_bottom_inverse()
b a
a b

sage: p = iet.Permutation('a b', 'b a', reduced=True)
sage: p.top_bottom_inverse() == p
True

sage: p = iet.Permutation('a b c d', 'c d a b')
sage: p.top_bottom_inverse()
c d a b
a b c d
\end{verbatim}
\texttt{vertical\_inverse()} 

Returns the left-right inverse.

You can also use the shorter \texttt{lr\_inverse()}

OUTPUT:

– a permutation

EXAMPLES:

\begin{verbatim}
sage: p = iet.Permutation('a b c','c a b')
sage: p.left_right_inverse()
c b a
b a c
sage: p = iet.Permutation('a b c d','c d a b')
sage: p.left_right_inverse()
d c b a
b a d c
sage: p = iet.GeneralizedPermutation('a a','b b c c')
sage: p.left_right_inverse()
a a
c c b b
sage: p = iet.Permutation('a b c','c b a',reduced=True)
sage: p.left_right_inverse() == p
True
sage: p = iet.Permutation('a b c','c a b',reduced=True)
sage: q = p.left_right_inverse()
sage: q == p
False
sage: q
a b c
b c a
sage: p = iet.GeneralizedPermutation('a a','b b c c',reduced=True)
sage: p.left_right_inverse() == p
True
sage: p = iet.GeneralizedPermutation('a b b','c c a',reduced=True)
sage: q = p.left_right_inverse()
sage: q == p
False
sage: q
a a b
b c c
\end{verbatim}

\section*{Warning} Internal class! Do not use directly!

\textbf{AUTHOR:}

• Vincent Delecroix (2008-12-20): initial version
arf_invariant()  
Returns the Arf invariant of the suspension of self.

OUTPUT:
integer – 0 or 1

EXAMPLES:

Permutations from the odd and even component of H(2,2):

```sage
a = range(10)
sage: b1 = [3,2,4,6,5,7,9,8,1,0]
sage: b0 = [6,5,4,3,2,7,9,8,1,0]
sage: p1 = iet.Permutation(a,b1)
sage: p1.arf_invariant()  
1
sage: p0 = iet.Permutation(a,b0)
sage: p0.arf_invariant()  
0
```

Permutations from the odd and even component of H(4,4):

```sage
a = range(11)
sage: b1 = [3,2,5,4,6,8,7,10,9,1,0]
sage: b0 = [5,4,3,2,6,8,7,10,9,1,0]
sage: p1 = iet.Permutation(a,b1)
sage: p1.arf_invariant()  
1
sage: p0 = iet.Permutation(a,b0)
sage: p0.arf_invariant()  
0
```

REFERENCES:

attached_in_degree()  
Returns the degree of the singularity at the right of the interval.

OUTPUT:
– a positive integer

EXAMPLES:

```sage
p1 = iet.Permutation('a b c d e f g','d c g f e b a')
sage: p2 = iet.Permutation('a b c d e f g','e d c g f b a')
sage: p1.attached_in_degree()  
1
sage: p2.attached_in_degree()  
3
```

attached_out_degree()  
Returns the degree of the singularity at the left of the interval.

OUTPUT:
– a positive integer

EXAMPLES:
attached_type()
Return the singularity degree attached on the left and the right.

OUTPUT:

([degree], angle_parity) – if the same singularity is attached on the left and right
([left_degree, right_degree], 0) – the degrees at the left and the right which are different

EXAMPLES:

With two intervals:

```
sage: p = iet.Permutation('a b', 'b a')
sage: p.attached_type()
([0], 1)
```

With three intervals:

```
sage: p = iet.Permutation('a b c', 'b c a')
sage: p.attached_type()
([0], 1)
sage: p = iet.Permutation('a b c', 'c a b')
sage: p.attached_type()
([0], 1)
sage: p = iet.Permutation('a b c', 'c b a')
sage: p.attached_type()
([0, 0], 0)
```

With four intervals:

```
sage: p = iet.Permutation('1 2 3 4', '4 3 2 1')
sage: p.attached_type()
([2], 0)
```

connected_component (marked_separatrix='no')
Returns a connected components of a stratum.

EXAMPLES:

Permutations from the stratum H(6):

```
sage: a = range(8)
sage: b_hyp = [7,6,5,4,3,2,1,0]
sage: b_odd = [3,2,5,4,7,6,1,0]
sage: b_even = [5,4,3,2,7,6,1,0]
sage: p_hyp = iet.Permutation(a, b_hyp)
sage: p_odd = iet.Permutation(a, b_odd)
sage: p_even = iet.Permutation(a, b_even)
```
Permutations from the stratum $H(4,4)$:

```python
sage: a = range(11)
sage: b_hyp = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
sage: b_odd = [3, 2, 5, 4, 6, 8, 7, 10, 9, 1, 0]
sage: b_even = [5, 4, 3, 2, 6, 8, 7, 10, 9, 1, 0]
sage: p_hyp = iet.Permutation(a, b_hyp)
sage: p_odd = iet.Permutation(a, b_odd)
sage: p_even = iet.Permutation(a, b_even)
```

```python
sage: p_hyp.connected_component() == AbelianStratum(4,4)
True
sage: p_hyp.connected_component() == H_hyp(4, 4)
```

As for stratum you can specify that you want to attach the singularity on the left of the interval using the option `marked_separatrix`:

```python
sage: a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: b4_odd = [4, 3, 6, 5, 7, 9, 8, 2, 1]
sage: b4_even = [6, 5, 4, 3, 7, 9, 8, 2, 1]
sage: b2_odd = [4, 3, 5, 7, 6, 9, 8, 2, 1]
sage: b2_even = [7, 6, 5, 4, 3, 9, 8, 2, 1]
sage: p4_odd = iet.Permutation(a, b4_odd)
sage: p4_even = iet.Permutation(a, b4_even)
sage: p2_odd = iet.Permutation(a, b2_odd)
sage: p2_even = iet.Permutation(a, b2_even)
sage: p4_odd.connected_component(marked_separatrix='out') == AbelianStratum(4,4)
True
```

```python
sage: p4_odd.connected_component(marked_separatrix='out') == H_odd(4, 4)
```

```python
sage: p2_odd.connected_component() == p4_odd.connected_component()
True
sage: p2_odd.connected_component('out') == p4_odd.connected_component('out')
```

\texttt{cylindric}()

Returns a permutation in the Rauzy class such that
twin[0][-1] == 0 twin[1][-1] == 0

**decompose()**

Returns the decomposition of self.

**OUTPUT:**

– a list of permutations

**EXAMPLES:**

```python
sage: p = iet.Permutation('a b c', 'c b a').decompose()[0]
sage: p
a b c
b a c
c b a
```

```python
sage: p1, p2, p3 = iet.Permutation('a b c d e', 'b a c e d').decompose()
sage: p1
a b
b a
da e
e d
```

**erase_marked_points()**

Returns a permutation equivalent to self but without marked points.

**EXAMPLES:**

```python
sage: a = iet.Permutation('a b1 b2 c d', 'd c b1 b2 a')
sage: a.erase_marked_points()
a b1 c d
d c b1 a
```

**genus()**

Returns the genus corresponding to any suspension of the permutation.

**OUTPUT:**

– a positive integer

**EXAMPLES:**

```python
sage: p = iet.Permutation('a b c', 'c b a')
sage: p.genus()
1
```

```python
sage: p = iet.Permutation('a b c d', 'd c b a')
sage: p.genus()
2
```

**REFERENCES:** Veech

**intersection_matrix()**

Returns the intersection matrix.
This $d \times d$ antisymmetric matrix is given by the rule:

$$m_{ij} = \begin{cases} 
1 & i < j \text{ and } \pi(i) > \pi(j) \\
-1 & i > j \text{ and } \pi(i) < \pi(j) \\
0 & \text{else}
\end{cases}$$

**OUTPUT:**

- a matrix

**EXAMPLES:**

```python
sage: p = iet.Permutation('a b c d','d c b a')
sage: p.intersection_matrix()
[ 0 1 1 1]
[-1 0 1 1]
[-1 -1 0 1]
[-1 -1 -1 0]
```

```python
sage: p = iet.Permutation('1 2 3 4 5','5 3 2 4 1')
sage: p.intersection_matrix()
[ 0 1 1 1 1]
[-1 0 1 0 1]
[-1 -1 0 0 1]
[-1 0 0 0 1]
[-1 -1 -1 -1 0]
```

**is_cylindric()**

Returns True if the permutation is Rauzy_In.

A permutation is cylindric if 1 and n are exchanged.

**EXAMPLES:**

```python
sage: iet.Permutation('1 2 3','3 2 1').is_cylindric()
True
sage: iet.Permutation('1 2 3','2 1 3').is_cylindric()
False
```

**is_hyperelliptic()**

Returns True if the permutation is in the class of the symmetric permutations (with eventual marked points).

This is equivalent to say that the suspension lives in an hyperelliptic stratum of Abelian differentials $H_{hyp}(2g-2)$ or $H_{hyp}(g-1, g-1)$ with some marked points.

**EXAMPLES:**

```python
sage: iet.Permutation('a b c d','d c b a').is_hyperelliptic()
True
sage: iet.Permutation('0 1 2 3 4 5','5 2 1 4 3 0').is_hyperelliptic()
False
```

**REFERENCES:**


**is_irreducible** *(return_decomposition=False)*

Tests the irreducibility.

An abelian permutation \( p = (p_0, p_1) \) is reducible if: \( \text{set}(p_0[:i]) = \text{set}(p_1[:i]) \) for an \( i < \text{len}(p_0) \)

**OUTPUT:**

- a boolean

**EXAMPLES:**

```python
sage: p = iet.Permutation('a b c', 'c b a')
sage: p.is_irreducible()
True
sage: p = iet.Permutation('a b c', 'b a c')
sage: p.is_irreducible()
False
```

**order_of_rauzy_action** *(winner, side=None)*

Returns the order of the action of a Rauzy move.

**INPUT:**

- **winner** - string 'top' or 'bottom'
- **side** - string 'left' or 'right'

**OUTPUT:**

An integer corresponding to the order of the Rauzy action.

**EXAMPLES:**

```python
sage: p = iet.Permutation('a b c d', 'd a c b')
sage: p.order_of_rauzy_action('top', 'right')
3
sage: p.order_of_rauzy_action('bottom', 'right')
2
sage: p.order_of_rauzy_action('top', 'left')
1
sage: p.order_of_rauzy_action('bottom', 'left')
3
```

**separatrix_diagram** *(side=False)*

Returns the separatrix diagram of the permutation.

**INPUT:**

- **side** - boolean

**OUTPUT:**

- a list of lists

**EXAMPLES:**

```python
sage: iet.Permutation([0, 1], [1, 0]).separatrix_diagram()
[[[0, 1], [0, 1]]]

sage: iet.Permutation('a b c d', 'd c b a').separatrix_diagram()
[[('d', 'a'), 'b', 'c', ('d', 'a'), 'b', 'c']]
```
stratum\(\text{marked\_separatrix} = \text{\textquoteleft}no\textquoteright\)\)

Returns the strata in which any suspension of this permutation lives.

OUTPUT:

• a stratum of Abelian differentials

EXAMPLES:

```python
sage: p = iet.Permutation('a b c', 'c b a')
sage: p.stratum()
doctest:warning
...
DeprecationWarning: AbelianStratum is deprecated and will be removed from Sage.
You are advised to install the surface_dynamics package via:
   sage -pip install surface_dynamics
If you do not have write access to the Sage installation you can alternatively do
   sage -pip install surface_dynamics --user
The package surface_dynamics subsumes all flat surface related computation that are currently available in Sage. See more information at
   http://www.labri.fr/perso/vdelecro/surface-dynamics/latest/
See http://trac.sagemath.org/20695 for details.
H(0, 0)

sage: p = iet.Permutation('a b c d', 'd a b c')
sage: p.stratum()
H(0, 0, 0)

sage: p = iet.Permutation(range(9), [8,5,2,7,4,1,6,3,0])
sage: p.stratum()
H(1, 1, 1, 1)
```

You can specify that you want to attach the singularity on the left (or on the right) with the option marked\_separatrix:

```python
sage: a = 'a b c d e f g h i j'
sage: b3 = 'd c g f e j i h b a'
sage: b2 = 'd c e g f j i h b a'
sage: b1 = 'e d c g f h j i b a'
sage: p3 = iet.Permutation(a, b3)
sage: p3.stratum()
H(3, 2, 1)
sage: p3.stratum(marked_separatrix='out')
H^\text{out}(3, 2, 1)
sage: p2 = iet.Permutation(a, b2)
sage: p2.stratum()
H(3, 2, 1)
sage: p2.stratum(marked_separatrix='out')
H^\text{out}(2, 3, 1)
sage: p1 = iet.Permutation(a, b1)
sage: p1.stratum()
H(3, 2, 1)
sage: p1.stratum(marked_separatrix='out')
H^\text{out}(1, 3, 2)
```

AUTHORS:
to_permutation()

Returns the permutation as an element of the symmetric group.

EXAMPLES:

```python
sage: p = iet.Permutation('a b c','c b a')
sage: p.to_permutation()
[3, 2, 1]
```

```python
sage: p = Permutation([2,4,1,3])
sage: q = iet.Permutation(p)
sage: q.to_permutation() == p
True
```

class sage.dynamics.interval_exchanges.template.PermutationLI

Bases: sage.dynamics.interval_exchanges.template.Permutation

Template for quadratic permutation.

**Warning:** Internal class! Do not use directly!

AUTHOR:

• Vincent Delecroix (2008-12-20): initial version

has_right_rauzy_move(winner)

Test of Rauzy movability (with an eventual specified choice of winner)

A quadratic (or generalized) permutation is rauzy_movable type depending on the possible length of the last interval. It's dependent of the length equation.

INPUT:

• winner - the integer 'top' or 'bottom'

EXAMPLES:

```python
sage: p = iet.GeneralizedPermutation('a a','b b')
sage: p.has_right_rauzy_move('top')
False
sage: p.has_right_rauzy_move('bottom')
False
```

```python
sage: p = iet.GeneralizedPermutation('a a b','b c c')
sage: p.has_right_rauzy_move('top')
True
sage: p.has_right_rauzy_move('bottom')
True
```

```python
sage: p = iet.GeneralizedPermutation('a a','b b c c')
sage: p.has_right_rauzy_move('top')
True
sage: p.has_right_rauzy_move('bottom')
False
```
**is_irreducible**(return_decomposition=False)

Test of reducibility

A quadratic (or generalized) permutation is **reducible** if there exists a decomposition

\[
A_1 u B_1 | \ldots | B_1 u A_2 \\
A_1 u B_2 | \ldots | B_2 u A_2
\]

where no corners is empty, or exactly one corner is empty and it is on the left, or two and they are both on the right or on the left. The definition is due to [BL2008] where they prove that the property of being irreducible is stable under Rauzy induction.

**INPUT:**

- return_decomposition - boolean (default: False) - if True, and the permutation is reducible, returns also the blocs \(A_1 u B_1, B_1 u A_2, A_1 u B_2\) and \(B_2 u A_2\) of a decomposition as above.

**OUTPUT:**

If return_decomposition is True, returns a 2-uple (test,decomposition) where test is the preceding test and decomposition is a 4-uple \((A_11,A_12,A_21,A_22)\) where:

\[
A_{11} = A_1 u B_1 \quad A_{12} = B_1 u A_2 \quad A_{21} = A_1 u B_2 \quad A_{22} = B_2 u A_2
\]

**EXAMPLES:**

```
sage: GP = iet.GeneralizedPermutation
sage: GP('a a','b b').is_irreducible()
False
sage: GP('a a b','b c c').is_irreducible()
True
sage: GP('1 2 3 4 5 1','5 6 6 4 3 2').is_irreducible()
True
```

**AUTHORS:**

- Vincent Delecroix (2008-12-20)

``` python
class sage.dynamics.interval_exchanges.template.RauzyDiagram(p, 
    right_induction=True, 
    left_induction=False, 
    left_right_inversion=False, 
    top_bottom_inversion=False, 
    symmetric=False)
```

**Bases:** `sage.structure.sage_object.SageObject`

Template for Rauzy diagrams.

**Warning:** Internal class! Do not use directly!

**AUTHORS:**
class Path(parent, *data)

Bases: sage.structure.sage_object.SageObject

Path in Rauzy diagram.

A path in a Rauzy diagram corresponds to a subsimplex of the simplex of lengths. This correspondence is obtained via the Rauzy induction. To a idoc IET we can associate a unique path in a Rauzy diagram. This establishes a correspondence between infinite full path in Rauzy diagram and equivalence topologic class of IET.

append(edge_type)

Append an edge to the path.

EXAMPLES:

```python
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: g = r.path(p)
sage: g.append('top')
```

Composition of path.

composition(function, composition=None)

Compose an edges function on a path

INPUT:

- path - either a Path or a tuple describing a path
- function - function must be of the form
- composition - the composition function

AUTHOR:

- Vincent Delecroix (2009-09-29)

EXAMPLES:

```python
sage: p = iet.Permutation('a b','b a')
sage: r = p.rauzy_diagram()
sage: def f(i,t):
....: if t is None: return []
....: return [t]
sage: g = r.path(p)
sage: g.composition(f, list.__add__)
```

edge_types()

Returns the edge types of the path.

EXAMPLES:

```python
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: g = r.path(p, 0, 1)
sage: g.edge_types()
```
end()  
Returns the last vertex of the path.

EXAMPLES:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: gl = r.path(p, 't', 'b', 't')
sage: gl.end() == p
True
sage: g2 = r.path(p, 'b', 't', 'b')
sage: g2.end() == p
True
```

extend(path)  
Extends self with another path.

EXAMPLES:

```
sage: p = iet.Permutation('a b c d','d c b a')
sage: r = p.rauzy_diagram()
sage: g1 = r.path(p,'t','t')
sage: g2 = r.path(p.rauzy_move('t',iteration=2),'b','b')
sage: g = r.path(p,'t','t','b','b')
sage: g == g1 + g2
True
sage: g = copy(g1)
sage: g.extend(g2)
sage: g == g1 + g2
True
```

is_loop()  
Tests whether the path is a loop (start point = end point).

EXAMPLES:

```
sage: p = iet.Permutation('a b','b a')
sage: r = p.rauzy_diagram()
sage: r.path(p).is_loop()
True
sage: r.path(p,0,1,0,0).is_loop()
True
```

losers()  
Returns a list of the loosers on the path.

EXAMPLES:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: g0 = r.path(p,'t','b','t')
sage: g0.losers()
['a', 'c', 'b']
sage: g1 = r.path(p,'b','t','b')
sage: g1.losers()
['c', 'a', 'b']
```

pop()  
Pops the queue of the path
a path corresponding to the last edge

**EXAMPLES:**

```python
sage: p = iet.Permutation('a b', 'b a')
sage: r = p.rauzy_diagram()
sage: g = r.path(p, 0, 1, 0)
sage: g0, g1, g2, g3 = g[0], g[1], g[2], g[3]
sage: g.pop() == r.path(g2, 0)
True
sage: g == r.path(g0, 0, 1)
True
sage: g.pop() == r.path(g1, 1)
True
sage: g == r.path(g0, 0)
True
sage: g == r.path(g0)
True
```
**cardinality()**
Returns the number of permutations in this Rauzy diagram.

**OUTPUT:**
- `integer` - the number of vertices in the diagram

**EXAMPLES:**
```python
sage: r = iet.RauzyDiagram('a b', 'b a')
sage: r.cardinality()
1
sage: r = iet.RauzyDiagram('a b c', 'c b a')
sage: r.cardinality()
3
sage: r = iet.RauzyDiagram('a b c d', 'd c b a')
sage: r.cardinality()
7
```

**complete**(p)
Completion of the Rauzy diagram.

Add to the Rauzy diagram all permutations that are obtained by successive operations defined by `edge_types()`. The permutation must be of the same type and the same length as the one used for the creation.

**INPUT:**
- `p` - a permutation of Interval exchange transformation

**Rauzy diagram** is the reunion of all permutations that could be obtained with successive rauzy moves. This function just use the functions `__getitem__` and has_rauzy_move and `rauzy_move` which must be defined for child and their corresponding permutation types.

**edge_iterator()**
Returns an iterator over the edges of the graph.

**EXAMPLES:**
```python
sage: p = iet.Permutation('a b', 'b a')
sage: r = p.rauzy_diagram()
sage: for e in r.edge_iterator():
      print(e[0].str(sep='/') + ' --> ' + e[1].str(sep='/'))
a b/b a --> a b/b a
a b/b a --> a b/b a
```

**edge_to_loser**(p=None, `edge_type=None`)
Return the corresponding loser

**edge_to_matrix**(p=None, `edge_type=None`)
Return the corresponding matrix

**INPUT:**
- `p` - a permutation
- `edge_type` - 0 or 1 corresponding to the type of the edge

**OUTPUT:**
A matrix

**EXAMPLES:**
```python
```
```python
sage: p = iet.Permutation('a b c', 'c b a')
sage: d = p.rauzy_diagram()
sage: d.edge_to_matrix(p, 1)
[1 0 1]
[0 1 0]
[0 0 1]
```

**edge_to_winner**(p=None, edge_type=None)

Return the corresponding winner

**edge_types**()

Print information about edges.

**EXAMPLES:**

```python
sage: r = iet.RauzyDiagram('a b', 'b a')
sage: r.edge_types()
0: rauzy_move(0, -1)
1: rauzy_move(1, -1)
```

```python
sage: r = iet.RauzyDiagram('a b', 'b a', left_induction=True)
sage: r.edge_types()
0: rauzy_move(0, -1)
1: rauzy_move(1, -1)
2: rauzy_move(0, 0)
3: rauzy_move(1, 0)
```

```python
sage: r = iet.RauzyDiagram('a b', 'b a', symmetric=True)
sage: r.edge_types()
0: rauzy_move(0, -1)
1: rauzy_move(1, -1)
2: symmetric()
```

**edge_types_index**(data)

Try to convert the data as an edge type.

**INPUT:**

• data - a string

**OUTPUT:**

integer

**EXAMPLES:**

For a standard Rauzy diagram (only right induction) the 0 index corresponds to the ‘top’ induction and the index 1 corresponds to the ‘bottom’ one:

```python
sage: p = iet.Permutation('a b c', 'c b a')
sage: r = p.rauzy_diagram()
sage: r.edge_types_index('top')
0
sage: r[p][0] == p.rauzy_move('top')
True
sage: r.edge_types_index('bottom')
1
sage: r[p][1] == p.rauzy_move('bottom')
True
```
The special operations (inversion and symmetry) always appears after the different Rauzy inductions:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram(symmetric=True)
```

```
sage: r.edge_types_index('symmetric')
2
sage: r[p][2] == p.symmetric()
True
```

This function always try to resolve conflictuous name. If it’s impossible a ValueError is raised:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram(left_induction=True)
```

```Traceback (most recent call last):
  ...  
ValueError: left and right inductions must be differentiated
```

```
sage: r.edge_types_index('top_right')
0
sage: r[p][0] == p.rauzy_move(0)
True
```

```
sage: r.edge_types_index('bottom_left')
3
sage: r[p][3] == p.rauzy_move('bottom', 'left')
True
```

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram(left_right_inversion=True,top_bottom_inversion=True)
```

```
Traceback (most recent call last):
  ...  
ValueError: left-right and top-bottom inversions must be differentiated
```

```
sage: r.edge_types_index('inversion')
1
```

```
sage: r.edge_types_index('lr_inverse')
2
```

```
sage: p.lr_inverse() == r[p][2]
True
```

```
sage: r.edge_types_index('tb_inverse')
3
```

```
sage: p.tb_inverse() == r[p][3]
True
```

Short names are accepted:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram(right_induction='top',top_bottom_inversion=True)
```

```
sage: r.edge_types_index('t')
0
sage: r.edge_types_index('tb')
1
```

```
sage: r.edge_types_index('inversion')
1
sage: r.edge_types_index('inverse')
1
sage: r.edge_types_index('i')
1
```
edges (labels=True)
Returns a list of the edges.

EXAMPLES:

```
sage: r = iet.RauzyDiagram('a b', 'b a')
sage: len(r.edges())
2
```

graph()
Returns the Rauzy diagram as a Graph object
The graph returned is more precisely a DiGraph (directed graph) with loops and multiedges allowed.

EXAMPLES:

```
sage: r = iet.RauzyDiagram('a b c', 'c b a')
sage: r
Rauzy diagram with 3 permutations
sage: r.graph()
Looped multi-digraph on 3 vertices
```

letters()
Returns the letters used by the RauzyDiagram.

EXAMPLES:

```
sage: r = iet.RauzyDiagram('a b', 'b a')
sage: r.alphabet()
{'a', 'b'}
sage: r.letters()  
['a', 'b']
sage: r.alphabet('ABCDEF')
sage: r.alphabet()
{'A', 'B', 'C', 'D', 'E', 'F'}
sage: r.letters()
['A', 'B']
```

path (*data)
Returns a path over this Rauzy diagram.

INPUT:

• initial_vertex - the initial vertex (starting point of the path)
• data - a sequence of edges

EXAMPLES:

```
sage: p = iet.Permutation('a b c', 'c b a')
sage: r = p.rauzy_diagram()
sage: g = r.path(p, 'top', 'bottom')
```

vertex_iterator()
Returns an iterator over the vertices

EXAMPLES:

```
sage: r = iet.RauzyDiagram('a b', 'b a')
sage: for p in r.vertex_iterator(): print(p)
```
a b
c d

sage: r = iet.RauzyDiagram('a b c d','d c b a')
sage: from six.moves import filter
sage: r_1n = filter(lambda x: x.is_cylindric(), r)
sage: for p in r_1n: print(p)
a b c d
c b a d

vertices()

Returns a list of the vertices.

EXAMPLES:

sage: r = iet.RauzyDiagram('a b','b a')
sage: for p in r.vertices(): print(p)
a b
b a

sage.dynamics.interval_exchanges.template.interval_conversion(interval=None)

Converts the argument in 0 or 1.

INPUT:

• winner - ‘top’ (or ‘t’ or 0) or bottom (or ‘b’ or 1)

OUTPUT:

integer – 0 or 1

sage: from sage.dynamics.interval_exchanges.template import interval_conversion
sage: interval_conversion('top')
0
sage: interval_conversion('t')
0
sage: interval_conversion(0)
0
sage: interval_conversion('bottom')
1
sage: interval_conversion('b')
1
sage: interval_conversion(1)
1

sage.dynamics.interval_exchanges.template.labelize_flip(couple)

Returns a string from a 2-uple couple of the form (name, flip).

sage.dynamics.interval_exchanges.template.side_conversion(side=None)

Converts the argument in 0 or -1.

INPUT:

• side - either ‘left’ (or ‘l’ or 0) or ‘right’ (or ‘r’ or -1)

OUTPUT:

integer – 0 or -1
sage: from sage.dynamics.interval_exchanges.template import side_conversion
sage: side_conversion('left')
0
sage: side_conversion('l')
0
sage: side_conversion(0)
0
sage: side_conversion('right')
-1
sage: side_conversion('r')
-1
sage: side_conversion(1)
-1
sage: side_conversion(-1)
-1

sage.dynamics.interval_exchanges.template.twin_list_iet(a=None)
Returns the twin list of intervals.

The twin intervals is the correspondance between positions of labels in such way that a[interval][position] is a[1-interval][twin[interval][position]]

INPUT:
• a - two lists of labels

OUTPUT:
list – a list of two lists of integers

sage.dynamics.interval_exchanges.template.twin_list_li(a=None)
Returns the twin list of intervals

INPUT:
• a - two lists of labels

OUTPUT:
list – a list of two lists of couples of integers

4.5 Interval Exchange Transformations and Linear Involution

**Warning:** This module is deprecated. You are advised to install and use the surface_dynamics package instead available at https://pypi.python.org/pypi/surface_dynamics/

An interval exchange transformation is a map defined on an interval (see help(iet.IntervalExchangeTransformation) for a more complete help.

**EXAMPLES:**

Initialization of a simple iet with integer lengths:

sage: T = iet.IntervalExchangeTransformation(Permutation([3,2,1]), [3,1,2])
doctest:warning
...
DeprecationWarning: IntervalExchangeTransformation is deprecated and will be removed from Sage.

(continues on next page)
You are advised to install the surface_dynamics package via:
sage -pip install surface_dynamics
If you do not have write access to the Sage installation you can alternatively do
sage -pip install surface_dynamics --user
The package surface_dynamics subsumes all flat surface related computation that are currently available in Sage. See more information at
http://www.labri.fr/perso/vdelecro/surface-dynamics/latest/
See http://trac.sagemath.org/20695 for details.
doctest:warning
...
DeprecationWarning: Permutation is deprecated and will be removed from Sage.
You are advised to install the surface_dynamics package via:
sage -pip install surface_dynamics
If you do not have write access to the Sage installation you can alternatively do
sage -pip install surface_dynamics --user
The package surface_dynamics subsumes all flat surface related computation that are currently available in Sage. See more information at
http://www.labri.fr/perso/vdelecro/surface-dynamics/latest/
See http://trac.sagemath.org/20695 for details.
sage: T
Interval exchange transformation of [0, 6] with permutation
1 2 3
3 2 1

Rotation corresponds to iet with two intervals:

sage: p = iet.Permutation('a b', 'b a')
sage: T = iet.IntervalExchangeTransformation(p, [1, (sqrt(5)-1)/2])
sage: print(T.in_which_interval(0))
a
sage: print(T.in_which_interval(T(0)))
a
sage: print(T.in_which_interval(T(T(0))))
b
sage: print(T.in_which_interval(T(T(T(0))))))
a

There are two plotting methods for iet:

sage: p = iet.Permutation('a b c', 'c b a')
sage: T = iet.IntervalExchangeTransformation(p, [1, 2, 3])

class sage.dynamics.interval_exchanges.iet.IntervalExchangeTransformation (permutation=None, lengths=None)

Bases: sage.structure.sage_object.SageObject

Interval exchange transformation

INPUT:

• permutation - a permutation (LabelledPermutationIET)
• lengths - the list of lengths

EXAMPLES:
Direct initialization:

```python
sage: p = iet.IET(('a b c', 'c b a'), {'a': 1, 'b': 1, 'c': 1})
sage: p.permutation()
a b c
c b a
sage: p.lengths()
[1, 1, 1]
```

Initialization from a `iet.Permutation`:

```python
sage: perm = iet.Permutation('a b c', 'c b a')
sage: l = [0.5, 1, 1.2]
sage: t = iet.IET(perm, l)
sage: t.permutation() == perm
True
sage: t.lengths() == l
True
```

Initialization from a `Permutation`:

```python
sage: p = Permutation([3, 2, 1])
sage: iet.IET(p, [1, 1, 1])
```

If it is not possible to convert lengths to real values an error is raised:

```python
sage: iet.IntervalExchangeTransformation(('a b', 'b a'), ['e', 'f'])
Traceback (most recent call last):
...  
TypeError: unable to convert 'e' to a float
```

The value for the lengths must be positive:

```python
sage: iet.IET(('a b', 'b a'), [-1, -1])
Traceback (most recent call last):
...  
ValueError: lengths must be positive
```

domain_singularities()

Returns the list of singularities of \( T \)

OUTPUT:

- list – positive reals that corresponds to singularities in the top interval

EXAMPLES:

```python
sage: t = iet.IET(('a b', 'b a'), [1, sqrt(2)])
sage: t.domain_singularities()
[0, 1, sqrt(2) + 1]
```

in_which_interval(\(x\), interval=0)

Returns the letter for which \( x \) is in this interval.

INPUT:

- \( x \) - a positive number
• interval - (default: ‘top’) ‘top’ or ‘bottom’

OUTPUT:

label – a label corresponding to an interval

```python
sage: t = iet.IntervalExchangeTransformation(('a b c', 'c b a'), [1, 1, 1])
```

```python
t.in_which_interval(0)
'a'
```

```python
t.in_which_interval(0.3)
'a'
```

```python
t.in_which_interval(1)
'b'
```

```python
t.in_which_interval(1.9)
'b'
```

```python
t.in_which_interval(2)
'c'
```

```python
t.in_which_interval(2.1)
'c'
```

```python
t.in_which_interval(3)
Traceback (most recent call last):
...
ValueError: your value does not lie in [0;1[
```

inverse() – Returns the inverse iet.

OUTPUT:

iet – the inverse interval exchange transformation

EXAMPLES:

```python
sage: p = iet.Permutation("a b", "b a")
sage: s = iet.IET(p, [1, sqrt(2) - 1])
sage: t = s.inverse()
sage: t.permutation()
b a
```

```python
t.lengths()
[1, sqrt(2) - 1]
sage: t*s
```

```
Interval exchange transformation of [0, sqrt(2)[ with permutation
aa bb
```

We can verify with the method .is_identity():

```python
sage: p = iet.Permutation("a b c d", "d a c b")
sage: s = iet.IET(p, [1, sqrt(2), sqrt(3), sqrt(5)])
sage: (s * s.inverse()).is_identity()
True
```

```python
(s.inverse() * s).is_identity()
```

```
True
```

is_identity() – Returns True if self is the identity.

OUTPUT:

boolean – the answer
EXAMPLES:

```
sage: p = iet.Permutation("a b","b a")
sage: q = iet.Permutation("c d","d c")
sage: s = iet.IET(p, [1,5])
sage: t = iet.IET(q, [5,1])
sage: (s*t).is_identity()
True
sage: (t*s).is_identity()
True
```

### length()

Returns the total length of the interval.

**OUTPUT:**

real – the length of the interval

**EXAMPLES:**

```
sage: t = iet.IntervalExchangeTransformation(('a b','b a'),[1,1])
sage: t.length()
2
```

### lengths()

Returns the list of lengths associated to this iet.

**OUTPUT:**

list – the list of lengths of subinterval

**EXAMPLES:**

```
sage: p = iet.IntervalExchangeTransformation(('a b','b a'),[1,3])
sage: p.lengths()
[1, 3]
```

### normalize(total=1)

Returns a interval exchange transformation of normalized lengths.

The normalization consists in multiplying all lengths by a constant in such way that their sum is given by `total` (default is 1).

**INPUT:**

- `total` - (default: 1) The total length of the interval

**OUTPUT:**

iet – the normalized iet

**EXAMPLES:**

```
sage: t = iet.IntervalExchangeTransformation(('a b','b a'), [1,3])
sage: t.length()
4
sage: s = t.normalize(2)
sage: s.length()
2
sage: s.lengths()
[1/2, 3/2]
```
permutation()

Returns the permutation associated to this iet.

OUTPUT:

permutation – the permutation associated to this iet

EXAMPLES:

```
sage: perm = iet.Permutation('a b c','c b a')
sage: p = iet.IntervalExchangeTransformation(perm,(1,2,1))
sage: p.permutation() == perm
True
```

plot(position=(0, 0), vertical_alignment='center', horizontal_alignment='left', interval_height=0.1, labels_height=0.05, fontsize=14, labels=True, colors=None)

Returns a picture of the interval exchange transformation.

INPUT:

- position - a 2-uple of the position
- horizontal_alignment - left (default), center or right
- labels - boolean (default: True)
- fontsize - the size of the label

OUTPUT:

2d plot – a plot of the two intervals (domain and range)

EXAMPLES:

```
sage: t = iet.IntervalExchangeTransformation(('a b','b a'),[1,1])
sage: t.plot_two_intervals()
Graphics object consisting of 8 graphics primitives
```

plot_function(**d)

Return a plot of the interval exchange transformation as a function.

INPUT:

- Any option that is accepted by line2d

OUTPUT:

2d plot – a plot of the iet as a function

EXAMPLES:

```
sage: t = iet.IntervalExchangeTransformation(('a b c d','d a c b'),[1,1,1,1])
sage: t.plot_function(rgbcolor=(0,1,0))
Graphics object consisting of 4 graphics primitives
```

plot_two_intervals(position=(0, 0), vertical_alignment='center', horizontal_alignment='left', interval_height=0.1, labels_height=0.05, fontsize=14, labels=True, colors=None)

Returns a picture of the interval exchange transformation.

INPUT:

- position - a 2-uple of the position
- horizontal_alignment - left (default), center or right
• labels - boolean (default: True)
  • fontsize - the size of the label

OUTPUT:
2d plot – a plot of the two intervals (domain and range)

EXAMPLES:

```
sage: t = iet.IntervalExchangeTransformation(('a b', 'b a'), [1, 1])
sage: t.plot_two_intervals()
Graphics object consisting of 8 graphics primitives
```

`range_singularities()`
Returns the list of singularities of $T^{-1}$

OUTPUT:
list – real numbers that are singular for $T^{-1}$

EXAMPLES:

```
sage: t = iet.IET(('a b', 'b a'), [1, sqrt(2)])
sage: t.range_singularities()
[0, sqrt(2), sqrt(2) + 1]
```

`rauzy_move(side='right', iterations=1)`
Performs a Rauzy move.

INPUT:
• side - 'left' (or 'l' or 0) or 'right' (or 'r' or 1)
  • iterations - integer (default: 1) the number of iteration of Rauzy moves to perform

OUTPUT:
iet – the Rauzy move of self

EXAMPLES:

```
sage: phi = QQbar((sqrt(5)-1)/2)
sage: t1 = iet.IntervalExchangeTransformation(('a b', 'b a'), [1, phi])
sage: t2 = t1.rauzy_move().normalize(t1.length())
sage: l2 = t2.lengths()
sage: l1 = t1.lengths()
sage: l2[0] == l1[1] and l2[1] == l1[0]
True
```

`show()`
Shows a picture of the interval exchange transformation

EXAMPLES:

```
sage: phi = QQbar((sqrt(5)-1)/2)
sage: t = iet.IntervalExchangeTransformation(('a b', 'b a'), [1, phi])
sage: t.show()
```

`singularities()`
The list of singularities of $T$ and $T^{-1}$.

OUTPUT:
list – two lists of positive numbers which corresponds to extremities of subintervals

EXAMPLES:

```
sage: t = iet.IntervalExchangeTransformation(('a b','b a'),[1/2,3/2])
sage: t.singularities()
[[0, 1/2, 2], [0, 3/2, 2]]
```
Functions and classes for mathematical sandpiles.

Version: 2.4

AUTHOR:

• David Perkinson (June 4, 2015) Upgraded from version 2.3 to 2.4.

MAJOR CHANGES

1. Eliminated dependence on 4ti2, substituting the use of Polyhedron methods. Thus, no optional packages are necessary.
2. Fixed bug in Sandpile.__init__ so that now multigraphs are handled correctly.
3. Created sandpiles to handle examples of Sandpiles in analogy with graphs, simplicial complexes, and polytopes. In the process, we implemented a much faster way of producing the sandpile grid graph.
4. Added support for open and closed sandpile Markov chains.
5. Added support for Weierstrass points.
6. Implemented the Cori-Le Borgne algorithm for computing ranks of divisors on complete graphs.

NEW METHODS

Sandpile: avalanche_polynomial, genus, group_gens, help, jacobian_representatives, markov_chain, picard_representatives, smith_form, stable_configs, stationary_density, tutte_polynomial.

SandpileConfig: burst_size, help.

SandpileDivisor: help, is_linearly_equivalent, is_q_reduced, is_weierstrass_pt, polytope, polytope_integer_pts, q_reduced, rank, simulate_threshold, stabilize, weierstrass_div, weierstrass_gap_seq, weierstrass_pts, weierstrass_rank_seq.

MINOR CHANGES

• The sink argument to Sandpile.__init__ now defaults to the first vertex.
• A SandpileConfig or SandpileDivisor may now be multiplied by an integer.
• Sped up __add__ method for SandpileConfig and SandpileDivisor.
• Enhanced string representation of a Sandpile (via __repr__ and the name methods).
• Recurrents for complete graphs and cycle graphs are computed more quickly.
• The stabilization code for SandpileConfig has been made more efficient.
• Added optional probability distribution arguments to add_random methods.
• Marshall Hampton (2010-1-10) modified for inclusion as a module within Sage library.

• David Perkinson (2010-12-14) added show3d(), fixed bug in resolution(), replaced elementary_divisors() with invariant_factors(), added show() for SandpileConfig and SandpileDivisor.

• David Perkinson (2010-9-18): removed is_undirected, added show(), added verbose arguments to several functions to display SandpileConfigs and divisors as lists of integers

• David Perkinson (2010-12-19): created separate SandpileConfig, SandpileDivisor, and Sandpile classes

• David Perkinson (2009-07-15): switched to using config_to_list instead of .values(), thus fixing a few bugs when not using integer labels for vertices.

• David Perkinson (2009): many undocumented improvements

• David Perkinson (2008-12-27): initial version

EXAMPLES:
For general help, enter Sandpile.help(), SandpileConfig.help(), and SandpileDivisor.help(). Miscellaneous examples appear below.

A weighted directed graph given as a Python dictionary:

```
sage: from sage.sandpiles import *
sage: g = {0: {}, 1: {0: 1, 2: 1, 3: 1}, ...
       2: {1: 1, 3: 1, 4: 1}, 3: {1: 1, 2: 1, 4: 1}, ...
       4: {2: 1, 3: 1}}
```

The associated sandpile with 0 chosen as the sink:

```
sage: S = Sandpile(g,0)
```

Or just:

```
sage: S = Sandpile(g)
```

A picture of the graph:

```
sage: S.show()  # long time
```

The relevant Laplacian matrices:

```
sage: S.laplacian()
[ 0 0 0 0 0]
[-1 3 -1 -1 0]
[ 0 -1 3 -1 -1]
[ 0 -1 -1 3 -1]
[ 0 0 -1 -1 2]
sage: S.reduced_laplacian()
[ 3 -1 -1 0]
[-1 3 -1 -1]
[-1 -1 3 -1]
[ 0 -1 -1 2]
```

The number of elements of the sandpile group for S:

```
sage: S.group_order()
8
```

The structure of the sandpile group:
sage: S.invariant_factors()
[1, 1, 1, 8]

The elements of the sandpile group for S:

sage: S.recurrents()
[{1: 2, 2: 2, 3: 2, 4: 1},
 {1: 2, 2: 2, 3: 2, 4: 0},
 {1: 2, 2: 1, 3: 2, 4: 0},
 {1: 2, 2: 2, 3: 0, 4: 1},
 {1: 2, 2: 0, 3: 2, 4: 1},
 {1: 2, 2: 2, 3: 1, 4: 0},
 {1: 2, 2: 1, 3: 2, 4: 1},
 {1: 2, 2: 2, 3: 1, 4: 1}]

The maximal stable element (2 grains of sand on vertices 1, 2, and 3, and 1 grain of sand on vertex 4:

sage: S.max_stable()
{1: 2, 2: 2, 3: 2, 4: 1}
sage: S.max_stable().values()
[2, 2, 2, 1]

The identity of the sandpile group for S:

sage: S.identity()
{1: 2, 2: 2, 3: 2, 4: 0}

An arbitrary sandpile configuration:

sage: c = SandpileConfig(S,[1,0,4,-3])
sage: c.equivalent_recurrent()
{1: 2, 2: 2, 3: 2, 4: 0}

Some group operations:

sage: m = S.max_stable()
sage: i = S.identity()
sage: m.values()
[2, 2, 2, 1]
sage: i.values()
[2, 2, 2, 0]
sage: m + i  # coordinate-wise sum
{1: 4, 2: 4, 3: 4, 4: 1}
sage: m - i
{1: 0, 2: 0, 3: 0, 4: 1}
sage: m & i  # add, then stabilize
{1: 2, 2: 2, 3: 2, 4: 1}
sage: e = m + m
sage: e
{1: 4, 2: 4, 3: 4, 4: 2}
sage: ~e  # stabilize
{1: 2, 2: 2, 3: 2, 4: 0}
sage: a = -m
sage: a & m
{1: 0, 2: 0, 3: 0, 4: 0}
sage: a * m  # add, then find the equivalent recurrent
{1: 2, 2: 2, 3: 2, 4: 0}
(continues on next page)
Firing an unstable vertex returns resulting configuration:

```
sage: c = S.max_stable() + S.identity()
sage: c.fire_vertex(1)
{1: 1, 2: 5, 3: 5, 4: 1}
sage: c
{1: 4, 2: 4, 3: 4, 4: 1}
```

Fire all unstable vertices:

```
sage: c.unstable()
[1, 2, 3]
sage: c.fire_unstable()
[1, 3, 2, 3, 3, 3, 4: 3]
```

Stabilize c, returning the resulting configuration and the firing vector:

```
sage: c.stabilize(True)
[(1: 2, 2: 2, 3: 2, 4: 1), (1: 6, 2: 8, 3: 8, 4: 8)]
sage: c
{1: 4, 2: 4, 3: 4, 4: 1}
sage: S.max_stable() & S.identity() == c.stabilize()
True
```

The number of superstable configurations of each degree:

```
sage: S.h_vector()
[1, 3, 4]
sage: S.postulation()
2
```

the saturated homogeneous toppling ideal:

```
sage: S.ideal()
Ideal (x1 - x0, x3*x2 - x0^2, x4^2 - x0^2, x2^3 - x4*x3*x0, x4*x2^2 - x3^2*x0, x3^3 - x4*x2*x0, x4*x3^2 - x2^2*x0) of Multivariate Polynomial Ring in x4, x3, x2, x1, x0 over Rational Field
```

its minimal free resolution:

```
sage: S.resolution()
'R^1 <-- R^7 <-- R^15 <-- R^13 <-- R^4'
```

and its Betti numbers:

```
sage: S.betti()
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
```

(continues on next page)

(continued from previous page)

Some various ways of creating Sandpiles:

```
sage: S = sandpiles.Complete(4) # for more options enter ``sandpile.TAB``
sage: S = sandpiles.Wheel(6)
```

A multidigraph with loops (vertices 0, 1, 2; for example, there is a directed edge from vertex 2 to vertex 1 of weight 3, which can be thought of as three directed edges of the form (2,3). There is also a single loop at vertex 2 and an edge (2,0) of weight 2):

```
sage: S = Sandpile({0:[1,2], 1:[0,0,2], 2:[0,0,1,1,1,2], 3:[2]})
```

Using the graph library (vertex 1 is specified as the sink; omitting this would make the sink vertex 0 by default):

```
sage: S = Sandpile(graphs.PetersenGraph(),1)
```

Distribution of avalanche sizes:

```
sage: S = sandpiles.Grid(10,10)
sage: m = S.max_stable()
sage: a = []
sage: for i in range(1000):
    ....: m = m.add_random()
    ....: m, f = m.stabilize(True)
    ....: a.append(sum(f.values()))
    ....:
sage: p = list_plot([[log(i+1),log(a.count(i))]
    for i in range max(a) if a.count(i)]
    sage: p.axes_labels(['log(N)','log(D(N))'])
    sage: t = text("Distribution of avalanche sizes", (2,2), rgbcolor=(1,0,0))
    sage: show(p+t,axes_labels=['log(N)','log(D(N))']) # long time
```

Working with sandpile divisors:

```
sage: S = sandpiles.Complete(4)
sage: D = SandpileDivisor(S, [0,0,0,5])
sage: E = D.stabilize(); E
{0: 1, 1: 1, 2: 1, 3: 2}
sage: D.is_linearly_equivalent(E)
True
sage: D.q_reduced()
{0: 4, 1: 0, 2: 0, 3: 1}
sage: S = sandpiles.Complete(4)
sage: D = SandpileDivisor(S, [0,0,0,5])
sage: E = D.stabilize(); E
{0: 1, 1: 1, 2: 1, 3: 2}
sage: D.is_linearly_equivalent(E)
True
sage: D.q_reduced()
{0: 4, 1: 0, 2: 0, 3: 1}
sage: D.rank()
2
sage: sorted(D.effective_div(), key=str)
```

(continues on next page)
class sage.sandpiles.sandpile.Sandpile(g, sink=None)

Bases: sage.graphs.digraph.DiGraph

Class for Dhar’s abelian sandpile model.

all_k_config(k)

The constant configuration with all values set to k.

INPUT:

k – integer

OUTPUT:

SandpileConfig

EXAMPLES:

sage: s = sandpiles.Diamond()
sage: s.all_k_config(7)
{1: 7, 2: 7, 3: 7}

all_k_div(k)

The divisor with all values set to k.

INPUT:

k – integer

OUTPUT:

SandpileDivisor

sage: sorted(D.effective_div(False))
[[0, 0, 0, 5], [0, 0, 4, 1], [0, 4, 0, 1], [1, 1, 1, 2], [4, 0, 0, 1]]

sage: D.rank()
2

sage: D.rank(True)
(2, {0: 2, 1: 1, 2: 0, 3: 0})

sage: E = D.rank(True)[1]  # E proves the rank is not 3

sage: E.values()
[2, 1, 0, 0]

sage: E.deg()
3

sage: rank(D - E)
-1

sage: (D - E).effective_div()
[]

sage: D.weierstrass_pts()
(0, 1, 2, 3)

sage: D.weierstrass_rank_seq(0)
(2, 1, 0, 0, 0, -1)

sage: D.weierstrass_pts()
(0, 1, 2, 3)

sage: D.weierstrass_rank_seq(0)
(2, 1, 0, 0, 0, -1)
EXAMPLES:

```
sage: S = sandpiles.House()
sage: S.all_k_div(7)
{0: 7, 1: 7, 2: 7, 3: 7, 4: 7}
```

**avalanche Polynomial** *(multi\_{variable}=True)*

The avalanche polynomial. See NOTE for details.

INPUT:

*multi\_{variable}*(default: True) boolean

OUTPUT:

polynomial

EXAMPLES:

```
sage: s = sandpiles.Complete(4)
sage: s.avalanche_polynomial()
9*x0*x1*x2 + 2*x0*x1 + 2*x0*x2 + 2*x1*x2 + 3*x0 + 3*x1 + 3*x2 + 24
sage: s.avalanche_polynomial(False)
9*x0^3 + 6*x0^2 + 9*x0 + 24
```

**Note:** For each nonsink vertex \( v \), let \( x_v \) be an indeterminate. If \((r, v)\) is a pair consisting of a recurrent \( r \) and nonsink vertex \( v \), then for each nonsink vertex \( w \), let \( n_w \) be the number of times vertex \( w \) fires in the stabilization of \( r + v \). Let \( M(r, v) \) be the monomial \( \prod_{w} x_w^{n_w} \), i.e., the exponent records the vector of \( n_w \) as \( w \) ranges over the nonsink vertices. The avalanche polynomial is then the sum of \( M(r, v) \) as \( r \) ranges over the recurrents and \( v \) ranges over the nonsink vertices. If multi\_{variable} is False, then set all the indeterminates equal to each other (and, thus, only count the number of vertex firings in the stabilizations, forgetting which particular vertices fired).

**betti** *(verbose=True)*

The Betti table for the homogeneous toppling ideal. If verbose is True, it prints the standard Betti table, otherwise, it returns a less formatted table.

INPUT:

*verbose*(default: True) boolean

OUTPUT:

Betti numbers for the sandpile

EXAMPLES:

```
sage: S = sandpiles.Diamond()
sage: S.betti()

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 :</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 :</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 :</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

sage: S.betti(False)
[1, 6, 9, 4]
```
**betti_complexes()**

The support-complexes with non-trivial homology. (See NOTE.)

**OUTPUT:**

list (of pairs [divisors, corresponding simplicial complex])

**EXAMPLES:**

```python
sage: S = Sandpile({0:{},1:{0: 1, 2: 1, 3: 4},2:{3: 5},3:{1: 1, 2: 1}},0)
sage: p = S.betti_complexes()
sage: p[0]
{(0: -8, 1: 5, 2: 4, 3: 1), Simplicial complex with vertex set (1, 2, 3) and facets {(3,), (1, 2)}]
sage: S.resolution()
'R^1 <-- R^5 <-- R^5 <-- R^1'
sage: S.betti()
0 1 2 3
-------------------------------
0: 1 - - -
1: - 5 5 -
2: - - - 1
-------------------------------
total: 1 5 5 1
sage: len(p)
11
sage: p[0][1].homology()
{0: 2, 1: 0}
sage: p[-1][1].homology()
{0: 0, 1: 0, 2: 2}
```

**Note:** A support-complex is the simplicial complex formed from the supports of the divisors in a linear system.

**burning_config()**

The minimal burning configuration.

**OUTPUT:**

dict (configuration)

**EXAMPLES:**

```python
sage: g = {0:{},1:{0:1,3:1,4:1},2:{0:1,3:1,5:1}, 3:{2:1,5:1},4:{1:1,3:1},5:{2:1,3:1}}
sage: S = Sandpile(g,0)
sage: S.burning_config()
{1: 2, 2: 0, 3: 1, 4: 1, 5: 0}
sage: S.burning_config().values()
[2, 0, 1, 1, 0]
sage: S.burning_script()
{1: 1, 2: 3, 3: 5, 4: 1, 5: 4}
sage: script = S.burning_script().values()
sage: script
[1, 3, 5, 1, 4]
sage: matrix(script)*S.reduced_laplacian()
[2 0 1 1 0]
```
Note: The burning configuration and script are computed using a modified version of Speer’s script algorithm. This is a generalization to directed multigraphs of Dhar’s burning algorithm.

A burning configuration is a nonnegative integer-linear combination of the rows of the reduced Laplacian matrix having nonnegative entries and such that every vertex has a path from some vertex in its support. The corresponding burning script gives the integer-linear combination needed to obtain the burning configuration. So if $b$ is the burning configuration, $\sigma$ is its script, and $\tilde{L}$ is the reduced Laplacian, then $\sigma \cdot \tilde{L} = b$. The minimal burning configuration is the one with the minimal script (its components are no larger than the components of any other script for a burning configuration).

The following are equivalent for a configuration $c$ with burning configuration $b$ having script $\sigma$:

- $c$ is recurrent;
- $c + b$ stabilizes to $c$;
- the firing vector for the stabilization of $c + b$ is $\sigma$.

**burning_script()**

A script for the minimal burning configuration.

**OUTPUT:**
dict

**EXAMPLES:**

```python
sage: g = {0:{},1:{0:1,3:1,4:1},2:{0:1,3:1,5:1},
3:{2:1,5:1},4:{1:1,3:1},5:{2:1,3:1}}
sage: S = Sandpile(g,0)
sage: S.burning_config()
{1: 2, 2: 0, 3: 1, 4: 1, 5: 0}
sage: S.burning_config().values()
[2, 0, 1, 1, 0]
sage: S.burning_script()
{1: 1, 2: 3, 3: 5, 4: 1, 5: 4}
sage: script = S.burning_script().values()
sage: script
[1, 3, 5, 1, 4]
sage: matrix(script)*S.reduced_laplacian()
[2 0 1 1 0]
```

Note: The burning configuration and script are computed using a modified version of Speer’s script algorithm. This is a generalization to directed multigraphs of Dhar’s burning algorithm.

A burning configuration is a nonnegative integer-linear combination of the rows of the reduced Laplacian matrix having nonnegative entries and such that every vertex has a path from some vertex in its support. The corresponding burning script gives the integer-linear combination needed to obtain the burning configuration. So if $b$ is the burning configuration, $s$ is its script, and $L_{\text{red}}$ is the reduced Laplacian, then $s \cdot L_{\text{red}} = b$. The minimal burning configuration is the one with the minimal script (its components are no larger than the components of any other script for a burning configuration).

The following are equivalent for a configuration $c$ with burning configuration $b$ having script $s$:

- $c$ is recurrent;
- $c + b$ stabilizes to $c$;
- the firing vector for the stabilization of $c + b$ is $s$. 
**canonical_divisor()**

The canonical divisor. This is the divisor with $\deg(v) - 2$ grains of sand on each vertex (not counting loops). Only for undirected graphs.

**OUTPUT:**

SandpileDivisor

**EXAMPLES:**

```sage
S = sandpiles.Complete(4)
s = S.canonical_divisor()
s
```

```sage
s = Sandpile({0: [1, 1], 1: [0, 0, 1, 1, 1]}, 0)
s = s.canonical_divisor()  # loops are disregarded
```

**Warning:** The underlying graph must be undirected.

**dict()**

A dictionary of dictionaries representing a directed graph.

**OUTPUT:**

dict

**EXAMPLES:**

```sage
S = sandpiles.Diamond()
s = S.dict()
s
S.sink()
```

**genus()**

The genus: (# non-loop edges) - (# vertices) + 1. Only defined for undirected graphs.

**OUTPUT:**

integer

**EXAMPLES:**

```sage
sandpiles.Complete(4).genus()
sandpiles.Cycle(5).genus()
```

**groebner()**

A Groebner basis for the homogeneous toppling ideal. It is computed with respect to the standard sandpile ordering (see ring).

**OUTPUT:**

Groebner basis
EXAMPLES:

```python
sage: S = sandpiles.Diamond()
sage: S.groebner()
[x3*x2^2 - x1^2*x0, x2^3 - x3*x1*x0, x3*x1^2 - x2^2*x0, x1^3 - x3*x2*x0, x3^2
→ - x0^2, x2*x1 - x0^2]
```

**group_gens** *(verbose=True)*

A minimal list of generators for the sandpile group. If **verbose** is **False** then the generators are represented as lists of integers.

**INPUT:**

**verbose** – (default: **True**) boolean

**OUTPUT:**

list of SandpileConfig (or of lists of integers if **verbose** is **False**)

**EXAMPLES:**

```python
sage: s = sandpiles.Cycle(5)
sage: s.group_gens()
[{1: 1, 2: 1, 3: 1, 4: 0}]
sage: s.group_gens()[0].order()
5
sage: s = sandpiles.Complete(5)
sage: s.group_gens(False)
[[2, 2, 3, 2], [2, 3, 2, 2], [3, 2, 2, 2]]
sage: [i.order() for i in s.group_gens()]
[5, 5, 5]
sage: s.invariant_factors()
[1, 5, 5, 5]
```

**group_order**

The size of the sandpile group.

**OUTPUT:**

integer

**EXAMPLES:**

```python
sage: S = sandpiles.House()
sage: S.group_order()
11
```

**h_vector**

The number of superstable configurations in each degree. Equivalently, this is the list of first differences of the Hilbert function of the (homogeneous) toppling ideal.

**OUTPUT:**

list of nonnegative integers

**EXAMPLES:**

```python
sage: s = sandpiles.Grid(2,2)
sage: s.hilbert_function()
[1, 5, 15, 35, 66, 106, 146, 178, 192]
sage: s.h_vector()
[1, 4, 10, 20, 31, 40, 40, 32, 14]
```
static help (verbose=True)

List of Sandpile-specific methods (not inherited from Graph). If verbose, include short descriptions.

INPUT:

verbose – (default: True) boolean

OUTPUT:

printed string

EXAMPLES:

sage: Sandpile.help() # long time
For detailed help with any method FOO listed below, enter "Sandpile.FOO?" or enter "S.FOO?" for any Sandpile S.

all_k_config -- The constant configuration with all values set to k.
all_k_div -- The divisor with all values set to k.
avalance_polynomial -- The avalanche polynomial.
betti -- The Betti table for the homogeneous toppling ideal.
betti_complexes -- The support-complexes with non-trivial homology.
burning_config -- The minimal burning configuration.
burning_script -- A script for the minimal burning configuration.
canonical_divisor -- The canonical divisor.
dict -- A dictionary of dictionaries representing a directed graph.
genus -- The genus: (# non-loop edges) - (# vertices) + 1.
groebner -- A Groebner basis for the homogeneous toppling ideal.
group_gens -- A minimal list of generators for the sandpile group.
group_order -- The size of the sandpile group.
h_vector -- The number of superstable configurations in each degree.
help -- List of Sandpile-specific methods (not inherited from "Graph").
hilbert_function -- The Hilbert function of the homogeneous toppling ideal.
ideal -- The saturated homogeneous toppling ideal.
identity -- The identity configuration.
in_degree -- The in-degree of a vertex or a list of all in-degrees.
invariant_factors -- The invariant factors of the sandpile group.
is_undirected -- Is the underlying graph undirected?
jacobian_representatives -- Representatives for the elements of the Jacobian group.
laplacian -- The Laplacian matrix of the graph.
markov_chain -- The sandpile Markov chain for configurations or divisors.
max_stable -- The maximal stable configuration.
max_stable_div -- The maximal stable divisor.
max_superstables -- The maximal superstable configurations.
min_recurrences -- The minimal recurrent elements.
nonsink_vertices -- The nonsink vertices.
nonspecial_divisors -- The nonspecial divisors.
out_degree -- The out-degree of a vertex or a list of all out-degrees.
picard_representatives -- Representatives of the divisor classes of degree \( d \) in the Picard group.
points -- Generators for the multiplicative group of zeros of the sandpile ideal.
postulation -- The postulation number of the toppling ideal.
recurrents -- The recurrent configurations.
reduced_laplacian -- The reduced Laplacian matrix of the graph.
reorder_vertices -- A copy of the sandpile with vertex names permuted.
resolution -- A minimal free resolution of the homogeneous toppling ideal.
ring -- The ring containing the homogeneous toppling ideal.
show -- Draw the underlying graph.
show3d -- Draw the underlying graph.
sink -- The sink vertex.
smith_form -- The Smith normal form for the Laplacian.
solve -- Approximations of the complex affine zeros of the sandpile ideal.
stable_configs -- Generator for all stable configurations.
stationary_density -- The stationary density of the sandpile.
superstables -- The superstable configurations.
symmetric_recurrents -- The symmetric recurrent configurations.
tutte_polynomial -- The Tutte polynomial of the underlying graph.
unsaturated_ideal -- The unsaturated, homogeneous toppling ideal.
version -- The version number of Sage Sandpiles.
zero_config -- The all-zero configuration.
zero_div -- The all-zero divisor.

hilbert_function()
The Hilbert function of the homogeneous toppling ideal.

OUTPUT:
list of nonnegative integers

EXAMPLES:

```sage```
s = sandpiles.Wheel(5)
s.hilbert_function()
```
[1, 5, 15, 31, 45]
```

```sage```
s = sandpiles.Wheel(5)
s.h_vector()
```
[1, 4, 10, 16, 14]
```

ideal (gens=False)
The saturated homogeneous toppling ideal. If gens is True, the generators for the ideal are returned instead.

INPUT:
gens – (default: False) boolean

OUTPUT:
ideal or, optionally, the generators of an ideal

EXAMPLES:

```sage```
S = sandpiles.Diamond() S.ideal()
```
(continues on next page)
Ideal (x2*x1 - x0^2, x3^2 - x0^2, x1^3 - x3*x2*x0, x3*x1^2 - x2^2*x0, x2^3 -
→x3*x1*x0, x3*x2^2 - x1^2*x0) of Multivariate Polynomial Ring in x3, x2, x1, ...
→x0 over Rational Field

sage: S.ideal(True)
[x2*x1 - x0^2, x3^2 - x0^2, x1^3 - x3*x2*x0, x3*x1^2 - x2^2*x0, x2^3 -
→x3*x1*x0, x3*x2^2 - x1^2*x0]
sage: S.ideal().gens() # another way to get the generators
[x2*x1 - x0^2, x3^2 - x0^2, x1^3 - x3*x2*x0, x3*x1^2 - x2^2*x0, x2^3 -
→x3*x1*x0, x3*x2^2 - x1^2*x0]

**identity (verbose=True)**

The identity configuration. If verbose is False, the configuration are converted to a list of integers.

**INPUT:**

verbose – (default: True) boolean

**OUTPUT:**

SandpileConfig or a list of integers If verbose is False, the configuration are converted to a list of integers.

**EXAMPLES:**

```python
sage: s = sandpiles.Diamond()
sage: s.identity()
{1: 2, 2: 2, 3: 0}
sage: s.identity(False)
[2, 2, 0]
sage: s.identity() & s.max_stable() == s.max_stable()
True
```

**in_degree (v=None)**

The in-degree of a vertex or a list of all in-degrees.

**INPUT:**

v – (optional) vertex name

**OUTPUT:**

integer or dict

**EXAMPLES:**

```python
sage: s = sandpiles.House()
sage: s.in_degree()
{0: 2, 1: 2, 2: 3, 3: 3, 4: 2}
sage: s.in_degree(2)
3
```

**invariant_factors ()**

The invariant factors of the sandpile group.

**OUTPUT:**

list of integers

**EXAMPLES:**

```python
```
sage: s = sandpiles.Grid(2,2)
sage: s.invariant_factors()
[1, 1, 8, 24]

is_undirected()
Is the underlying graph undirected? True if $(u, v)$ is an edge if and only if $(v, u)$ is an edge, each edge with the same weight.

OUTPUT:
boolean

EXAMPLES:

sage: sandpiles.Complete(4).is_undirected()
True
sage: s = Sandpile({0:[1,2], 1:[0,2], 2:[0]}, 0)
sage: s.is_undirected()
False

jacobian_representatives (verbose=True)
Representatives for the elements of the Jacobian group. If verbose is False, then lists representing the divisors are returned.

INPUT:
verbose – (default: True) boolean

OUTPUT:
list of SandpileDivisor (or of lists representing divisors)

EXAMPLES:

For an undirected graph, divisors of the form $s - \deg(s) \cdot \text{sink}$ as $s$ varies over the superstables forms a distinct set of representatives for the Jacobian group:

```
sage: s = sandpiles.Complete(3)
sage: s.superstables(False)
[[0, 0], [0, 1], [1, 0]]
sage: s.jacobian_representatives(False)
[[0, 0, 0], [-1, 0, 1], [-1, 1, 0]]
```

If the graph is directed, the representatives described above may by equivalent modulo the rowspan of the Laplacian matrix:

```
sage: s = Sandpile({0: {1: 1, 2: 2}, 1: {0: 2, 2: 4}, 2: {0: 4, 1: 2}}, 0)
sage: s.group_order()
28
sage: s.jacobian_representatives()
[[-5, 3, 2], [-4, 3, 2]]
```

Let $\tau$ be the nonnegative generator of the kernel of the transpose of the Laplacian, and let $\tau_s$ be it sink component, then the sandpile group is isomorphic to the direct sum of the cyclic group of order $\tau_s$ and the Jacobian group. In the example above, we have:

```
sage: s.laplacian().left_kernel()
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[14 5 8]
```
Note: The Jacobian group is the set of all divisors of degree zero modulo the integer rowspan of the Laplacian matrix.

```
laplacian()
```

The Laplacian matrix of the graph. Its rows encode the vertex firing rules.

**OUTPUT:**

matrix

**EXAMPLES:**

```
sage: G = sandpiles.Diamond()
sage: G.laplacian()
[ 2 -1 -1  0]
[-1  3 -1 -1]
[-1 -1  3 -1]
[ 0 -1 -1  2]
```

**Warning:** The function `laplacian_matrix` should be avoided. It returns the indegree version of the Laplacian.

```
markov_chain(state, distrib=None)
```

The sandpile Markov chain for configurations or divisors. The chain starts at `state`. See NOTE for details.

**INPUT:**

- `state` – SandpileConfig, SandpileDivisor, or list representing one of these
- `distrib` – (optional) list of nonnegative numbers summing to 1 (representing a prob. dist.)

**OUTPUT:**

generator for Markov chain (see NOTE)

**EXAMPLES:**

```
sage: s = sandpiles.Complete(4)
sage: m = s.markov_chain([0,0,0])
sage: next(m)  # random
{1: 0, 2: 0, 3: 0}
sage: next(m).values()  # random
[0, 0, 0]
sage: next(m).values()  # random
[0, 0, 0]
sage: next(m).values()  # random
[0, 1, 0]
sage: next(m).values()  # random
[0, 2, 0]
sage: next(m).values()  # random
[0, 2, 1]
sage: next(m).values()  # random
[1, 2, 1]
sage: next(m).values()  # random
```

(continues on next page)
Note: The closed sandpile Markov chain has state space consisting of the configurations on a sandpile. It transitions from a state by choosing a vertex at random (according to the probability distribution $distrib$), dropping a grain of sand at that vertex, and stabilizing. If the chosen vertex is the sink, the chain stays at the current state.

The open sandpile Markov chain has state space consisting of the recurrent elements, i.e., the state space is the sandpile group. It transitions from the configuration $c$ by choosing a vertex $v$ at random according to $distrib$. The next state is the stabilization of $c + v$. If $v$ is the sink vertex, then the stabilization of $c + v$ is defined to be $c$.

Note that in either case, if $distrib$ is specified, its length is equal to the total number of vertices (including the sink).

REFERENCES:

- [Lev2014]

max_stable()  
The maximal stable configuration.

OUTPUT:

SandpileConfig (the maximal stable configuration)

EXAMPLES:

sage: S = sandpiles.House()  
sage: S.max_stable()  
{1: 1, 2: 2, 3: 2, 4: 1}

max_stable_div()  
The maximal stable divisor.

OUTPUT:

SandpileDivisor (the maximal stable divisor)
EXAMPLES:

```python
sage: s = sandpiles.Diamond()
sage: s.max_stable_div()
{0: 1, 1: 2, 2: 2, 3: 1}
sage: s.out_degree()
{0: 2, 1: 3, 2: 3, 3: 2}
```

**max_superstables** *(verbose=True)*

The maximal superstable configurations. If the underlying graph is undirected, these are the superstables of highest degree. If `verbose` is False, the configurations are converted to lists of integers.

**INPUT:**

`verbose` – (default: True) boolean

**OUTPUT:**

tuple of SandpileConfig

**EXAMPLES:**

```python
sage: s = sandpiles.Diamond()
sage: s.superstables(False)
[[0, 0, 0],
 [0, 0, 1],
 [1, 0, 1],
 [0, 2, 0],
 [2, 0, 0],
 [0, 1, 1],
 [1, 0, 0],
 [0, 1, 0]]
sage: s.max_superstables(False)
[[1, 0, 1], [0, 2, 0], [2, 0, 0], [0, 1, 1]]
sage: s.h_vector()
[1, 3, 4]
```

**min_recurrents** *(verbose=True)*

The minimal recurrent elements. If the underlying graph is undirected, these are the recurrent elements of least degree. If `verbose` is False, the configurations are converted to lists of integers.

**INPUT:**

`verbose` – (default: True) boolean

**OUTPUT:**

list of SandpileConfig

**EXAMPLES:**

```python
sage: s = sandpiles.Diamond()
sage: s.recurrents(False)
[[2, 2, 1],
 [2, 2, 0],
 [1, 2, 0],
 [2, 0, 1],
 [0, 2, 1],
 [2, 1, 0],
 [1, 2, 1],
 [2, 1, 1]]
```

(continues on next page)
nonsink_vertices()

The nonsink vertices.

OUTPUT:
list of vertices

EXAMPLES:

sage: s = sandpiles.Grid(2,3)
sage: s.nonsink_vertices()
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3)]

nonspecial_divisors(\texttt{verbose=\texttt{True}})

The nonspecial divisors. Only for undirected graphs. (See NOTE.)

INPUT:
\texttt{verbose} – (default: \texttt{True}) boolean

OUTPUT:
list (of divisors)

EXAMPLES:

sage: S = sandpiles.Complete(4)
sage: ns = S.nonspecial_divisors()
sage: D = ns[0]
sage: D.values()
[-1, 0, 1, 2]
sage: D.deg()
2
sage: [i.effective_div() for i in ns]
[[], [], [], [], [], []]

\textbf{Note:} The “nonspecial divisors” are those divisors of degree \(g - 1\) with empty linear system. The term is only defined for undirected graphs. Here, \(g = |E| - |V| + 1\) is the genus of the graph (not counting loops as part of \(|E|\)). If \texttt{verbose} is \texttt{False}, the divisors are converted to lists of integers.

\textbf{Warning:} The underlying graph must be undirected.

out_degree(\texttt{v=\texttt{None}})

The out-degree of a vertex or a list of all out-degrees.

INPUT:
\texttt{v} - (optional) vertex name

OUTPUT:
integer or dict
EXAMPLES:

```python
sage: s = sandpiles.House()
sage: s.out_degree()
{0: 2, 1: 2, 2: 3, 3: 3, 4: 2}
sage: s.out_degree(2)
3
```

**picard_representatives** *(d, verbose=True)*

Representatives of the divisor classes of degree *d* in the Picard group. (Also see the documentation for jacobian_representatives.)

**INPUT:**

- *d* – integer
- *verbose* – (default: True) boolean

**OUTPUT:**

list of SandpileDivisors (or lists representing divisors)

**EXAMPLES:**

```python
sage: s = sandpiles.Complete(3)
sage: s.superstables(False)
[[0, 0], [0, 1], [1, 0]]
sage: s.jacobian_representatives(False)
[[0, 0, 0], [-1, 0, 1], [-1, 1, 0]]
sage: s.picard_representatives(3,False)
[[3, 0, 0], [2, 0, 1], [2, 1, 0]]
```

**points()**

Generators for the multiplicative group of zeros of the sandpile ideal.

**OUTPUT:**

list of complex numbers

**EXAMPLES:**

The sandpile group in this example is cyclic, and hence there is a single generator for the group of solutions.

```python
sage: S = sandpiles.Complete(4)
sage: S.points()
[[1, I, -I], [I, 1, -I]]
```

**postulation()**

The postulation number of the toppling ideal. This is the largest weight of a superstable configuration of the graph.

**OUTPUT:**

nonnegative integer

**EXAMPLES:**

```python
sage: s = sandpiles.Complete(4)
sage: s.postulation()
3
```

**recurrents** *(verbose=True)*

The recurrent configurations. If *verbose* is False, the configurations are converted to lists of integers.
INPUT:

verbose – (default: True) boolean

OUTPUT:

list of recurrent configurations

EXAMPLES:

```
sage: r = Sandpile(graphs.HouseXGraph(),0).recurrents()
sage: r[:3]
{1: 2, 2: 3, 3: 3, 4: 1}, {1: 1, 2: 3, 3: 3, 4: 0}, {1: 1, 2: 3, 3: 3, 4: 1}
sage: sandpiles.Complete(4).recurrents(False)
[[2, 2, 2], [2, 2, 1], [2, 1, 2], [1, 2, 2], [2, 2, 0], [2, 0, 2], [0, 2, 2], [2, 1, 1], [1, 2, 1], [1, 1, 2], [2, 1, 0], [2, 0, 1], [1, 2, 0], [1, 0, 2], [0, 2, 1], [0, 1, 2]]
sage: sandpiles.Cycle(4).recurrents(False)
[[1, 1, 1], [0, 1, 1], [1, 0, 1], [1, 1, 0]]
```

reduced_laplacian()

The reduced Laplacian matrix of the graph.

OUTPUT:

matrix

EXAMPLES:

```
sage: S = sandpiles.Diamond()
sage: S.laplacian()
[ 2 -1 -1 0]
[-1 3 -1 -1]
[-1 -1 3 -1]
[ 0 -1 -1 2]
sage: S.reduced_laplacian()
[ 3 -1 -1]
[-1 3 -1]
[-1 -1 2]
```

Note: This is the Laplacian matrix with the row and column indexed by the sink vertex removed.

reorder_vertices()

A copy of the sandpile with vertex names permuted.

After reordering, vertex $u$ comes before vertex $v$ in the list of vertices if $u$ is closer to the sink.
OUTPU:
Sandpile

EXAMPLES:

```python
sage: S = Sandpile({0:[1], 2:[0,1], 1:[2]})
sage: S.dict()
{0: (1: 1), 1: (2: 1), 2: (0: 1, 1: 1)}
sage: T = S.reorder_vertices()
The vertices 1 and 2 have been swapped:
```
```python
sage: T.dict()
{0: (1: 1), 1: (0: 1, 2: 1), 2: (0: 1)}
```

`resolution(\texttt{verbose=\texttt{False}})`
A minimal free resolution of the homogeneous toppling ideal. If \texttt{verbose} is \texttt{True}, then all of the mappings are returned. Otherwise, the resolution is summarized.

INPUT:
\texttt{verbose} – (default: \texttt{False}) boolean

OUTPUT:
free resolution of the toppling ideal

EXAMPLES:

```python
sage: S = Sandpile({0: {}, 1: {0: 1, 2: 1, 3: 4}, 2: {3: 5}, 3: {1: 1, 2: 1}}, \rightarrow 0)
sage: S.resolution() # a Gorenstein sandpile graph
'R^1 <-- R^5 <-- R^5 <-- R^1'
sage: S.resolution(True)
[ [ x1^2 - x3*x0 x3*x1 - x2*x0 x3^2 - x2 *x1 x2 *x3 - x0^2 x2^2 - x1 *x0],
  [ x3 x2 0 x0 0] [ x2^2 - x1*x0]
[ -x1 -x3 x2 0 -x0] [-x2*x3 + x0^2]
[ x0 x1 0 x2 0] [-x3^2 + x2*x1]
[ 0 0 -x1 -x3 x2] [x3*x1 - x2*x0]
[ 0 0 x0 x1 -x3], [ x1^2 - x3*x0]
]
sage: r = S.resolution(True)
sage: r[0]*r[1]
[0 0 0 0 0]
sage: r[1]*r[2]
[0]
[0]
[0]
[0]

```

`ring()`
The ring containing the homogeneous toppling ideal.

OUTPUT:
ring

EXAMPLES:
sage: S = sandpiles.Diamond()
sage: S.ring()
Multivariate Polynomial Ring in x3, x2, x1, x0 over Rational Field
sage: S.ring().gens()
(x3, x2, x1, x0)

Note: The indeterminate $x_i$ corresponds to the $i$-th vertex as listed my the method vertices. The term-ordering is degrevlex with indeterminates ordered according to their distance from the sink (larger indeterminates are further from the sink).

**show(**kwds**)**
Draw the underlying graph.

**INPUT:**
kwds – (optional) arguments passed to the show method for Graph or DiGraph

**EXAMPLES:**

```sage
sage: S = Sandpile({0:[], 1:[0,3,4], 2:[0,3,5], 3:[2,5], 4:[1,1], 5:[2,4]})
sage: S.show()
sage: S.show(graph_border=True, edge_labels=True)
```

**show3d(**kwds**)**
Draw the underlying graph.

**INPUT:**
kwds – (optional) arguments passed to the show method for Graph or DiGraph

**EXAMPLES:**

```sage
sage: S = sandpiles.House()
sage: S.show3d() # long time
```

**sink()**
The sink vertex.

**OUTPUT:**
sink vertex

**EXAMPLES:**

```sage
sage: G = sandpiles.House()
sage: G.sink()
0
sage: H = sandpiles.Grid(2,2)
sage: H.sink()
(0, 0)
sage: type(H.sink())
<... 'tuple'>
```

**smith_form()**
The Smith normal form for the Laplacian. In detail: a list of integer matrices $D, U, V$ such that $ULV = D$ where $L$ is the transpose of the Laplacian, $D$ is diagonal, and $U$ and $V$ are invertible over the integers.

**OUTPUT:**
list of integer matrices

EXAMPLES:

```
sage: s = sandpiles.Complete(4)
sage: D,U,V = s.smith_form()
sage: D
[1 0 0 0]
[0 4 0 0]
[0 0 4 0]
[0 0 0 0]
sage: U*s.laplacian()*V == D  # Laplacian symmetric => transpose not necessary
True
```

`solve()`
Approximations of the complex affine zeros of the sandpile ideal.

OUTPUT:
list of complex numbers

EXAMPLES:

```
sage: S = Sandpile({0: {}, 1: {2: 2}, 2: {0: 4, 1: 1}}, 0)
sage: S.solve()
[[-0.707107 + 0.707107*I, 0.707107 - 0.707107*I], [-0.707107 - 0.707107*I, 0.707107 + 0.707107*I], [-I, -I], [I, I], [0.707107 + 0.707107*I, -0.707107 - 0.707107*I], [0.707107 - 0.707107*I, -0.707107 + 0.707107*I], [1, 1], [-1, -1]]
sage: len(_)
8
sage: S.group_order()
8
```

Note: The solutions form a multiplicative group isomorphic to the sandpile group. Generators for this
group are given exactly by `points()`.

`stable_configs(smax=None)`
Generator for all stable configurations. If `smax` is provided, then the generator gives all stable configurations
less than or equal to `smax`. If `smax` does not represent a stable configuration, then each component
of `smax` is replaced by the corresponding component of the maximal stable configuration.

INPUT:

`smax` – (optional) SandpileConfig or list representing a SandpileConfig

OUTPUT:

generator for all stable configurations

EXAMPLES:

```
sage: s = sandpiles.Complete(3)
sage: a = s.stable_configs()
sage: next(a)
{1: 0, 2: 0}
sage: [i.values() for i in a]
[[0, 1], [1, 0], [1, 1]]
sage: b = s.stable_configs([1,0])
```
The stationary density of a sandpile is the sum $\sum_c (\deg(c) + \deg(s))$ where $\deg(s)$ is the degree of the sink and the sum is over all recurrent configurations.

REFERENCES:

- [Lev2014]

superstables (verbose=True)

The superstable configurations. If verbose is False, the configurations are converted to lists of integers. Superstables for undirected graphs are also known as G-parking functions.

INPUT:

verbose – (default: True) boolean

OUTPUT:

list of SandpileConfig

EXAMPLES:

```
sage: sp = Sandpile(graphs.HouseXGraph(), 0).superstables()
sage: sp[:3]
[[0, 0, 0], [0, 0, 1], [0, 0, 0]]
sage: sp = sandpiles.Complete(4).superstables(False)
[[0, 0, 0], [0, 0, 0], [0, 0, 1], [0, 0, 0], [0, 0, 0], [0, 1, 1], [0, 1, 1], [0, 1, 0], [0, 1, 0], [0, 1, 0], [0, 1, 1], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

(continues on next page)
sage: sandpiles.Cycle(4).superstables(False)
[[0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]]

**symmetric_recurrents** *(orbits)*

The symmetric recurrent configurations.

**INPUT:**

orbits - list of lists partitioning the vertices

**OUTPUT:**

list of recurrent configurations

**EXAMPLES:**

```python
sage: S = Sandpile({0: {},
....: 1: {0: 1, 2: 1, 3: 1},
....: 2: {1: 1, 3: 1, 4: 1},
....: 3: {1: 1, 2: 1, 4: 1},
....: 4: {2: 1, 3: 1}})
sage: S.symmetric_recurrents([[1], [2, 3], [4]])
[{1: 2, 2: 2, 3: 2, 4: 1}, {1: 2, 2: 2, 3: 2, 4: 0}]
sage: S.recurrents()
[{1: 2, 2: 2, 3: 2, 4: 1}, {1: 2, 2: 2, 3: 2, 4: 0}, {1: 2, 2: 1, 3: 2, 4: 0}, {1: 2, 2: 2, 3: 1, 4: 0}, {1: 2, 2: 1, 3: 2, 4: 1}, {1: 2, 2: 2, 3: 1, 4: 1}]
```

**Note:** The user is responsible for ensuring that the list of orbits comes from a group of symmetries of the underlying graph.

**tutte_polynomial** ()

The Tutte polynomial of the underlying graph. Only defined for undirected sandpile graphs.

**OUTPUT:**

polynomial

**EXAMPLES:**

```python
sage: s = sandpiles.Complete(4)
sage: s.tutte_polynomial()
x^3 + y^3 + 3*x^2 + 4*x*y + 3*y^2 + 2*x + 2*y
sage: s.tutte_polynomial().subs(x=1)
y^3 + 3*y^2 + 6*y + 6
sage: s.tutte_polynomial().subs(x=1).coefficients() == s.h_vector()
True
```

**unsaturated_ideal** ()

The unsaturated, homogeneous toppling ideal.
ideal

EXAMPLES:

```python
sage: S = sandpiles.Diamond()
sage: S.unsaturated_ideal().gens()
[x1^3 - x3*x2*x0, x2^3 - x3*x1*x0, x3^2 - x2*x1]
sage: S.ideal().gens()
[x2*x1 - x0^2, x3^2 - x0^2, x1^3 - x3*x2*x0, x3*x1^2 - x2^2*x0, x2^3 - 
x3*x1*x0, x3*x2^2 - x1^2*x0]
```

static version()

The version number of Sage Sandpiles.

OUTPUT:

string

EXAMPLES:

```python
sage: Sandpile.version()
Sage Sandpiles Version 2.4
sage: S = sandpiles.Complete(3)
sage: S.version()
Sage Sandpiles Version 2.4
```

zero_config()

The all-zero configuration.

OUTPUT:

SandpileConfig

EXAMPLES:

```python
sage: s = sandpiles.Diamond()
sage: s.zero_config()
{1: 0, 2: 0, 3: 0}
```

zero_div()

The all-zero divisor.

OUTPUT:

SandpileDivisor

EXAMPLES:

```python
sage: S = sandpiles.House()
sage: S.zero_div()
{0: 0, 1: 0, 2: 0, 3: 0, 4: 0}
```

class sage.sandpiles.sandpile.SandpileConfig(S,c)

Bases: dict

Class for configurations on a sandpile.

**add_random** (**distrib=None**)

Add one grain of sand to a random vertex. Optionally, a probability distribution, distrib, may be placed on the vertices or the nonsink vertices. See NOTE for details.
INPUT:

distrib – (optional) list of nonnegative numbers summing to 1 (representing a prob. dist.)

OUTPUT:
SandpileConfig

EXAMPLES:

```sage
sage: s = sandpiles.Complete(4)
sage: c = s.zero_config()
sage: c.add_random()  # random
{1: 0, 2: 1, 3: 0}
sage: c
{1: 0, 2: 0, 3: 0}
sage: c.add_random([0.1,0.1,0.8])  # random
{1: 0, 2: 0, 3: 1}
sage: c.add_random([0.7,0.1,0.1,0.1])  # random
{1: 0, 2: 0, 3: 0}
```

We compute the “sizes” of the avalanches caused by adding random grains of sand to the maximal stable configuration on a grid graph. The function `stabilize()` returns the firing vector of the stabilization, a dictionary whose values say how many times each vertex fires in the stabilization.

```sage
sage: S = sandpiles.Grid(10,10)
sage: m = S.max_stable()
sage: a = []
sage: for i in range(1000):
...:     m = m.add_random()
...:     m, f = m.stabilize(True)
...:     a.append(sum(f.values()))
...:
sage: p = list_plot([[log(i+1),log(a.count(i))] for i in [0..max(a)] if a.count(i)])
sage: p.axes_labels(['log(N)','log(D(N))'])
sage: t = text("Distribution of avalanche sizes", (2,2), rgbcolor=(1,0,0))
sage: show(p+t,axes_labels=['log(N)','log(D(N))'])
```

Note: If `distrib` is `None`, then the probability is the uniform probability on the nonsink vertices. Otherwise, there are two possibilities:

(i) the length of `distrib` is equal to the number of vertices, and `distrib` represents a probability distribution on all of the vertices. In that case, the sink may be chosen at random, in which case, the configuration is unchanged.

(ii) Otherwise, the length of `distrib` must be equal to the number of nonsink vertices, and `distrib` represents a probability distribution on the nonsink vertices.

**Warning:** If `distrib != None`, the user is responsible for assuring the sum of its entries is 1 and that its length is equal to the number of sink vertices or the number of nonsink vertices.

`burst_size(v)`
The burst size of the configuration with respect to the given vertex.

INPUT:
v – vertex

OUTPUT:

integer

EXAMPLES:

```
sage: s = sandpiles.Diamond()
sage: [i.burst_size(0) for i in s.recurrents()]
[1, 1, 1, 1, 1, 1, 1, 1]
sage: [i.burst_size(1) for i in s.recurrents()]
[0, 0, 1, 2, 1, 2, 0, 2]
```

Note: To define \( c.burst(v) \), if \( v \) is not the sink, let \( c' \) be the unique recurrent for which the stabilization of \( c' + v \) is \( c \). The burst size is then the amount of sand that goes into the sink during this stabilization. If \( v \) is the sink, the burst size is defined to be 1.

REFERENCES:

- [Lev2014]

deg ()
The degree of the configuration.

OUTPUT:

integer

EXAMPLES:

```
sage: S = sandpiles.Complete(3)
sage: c = SandpileConfig(S, [1,2])
sage: c.deg()
3
```

dualize ()
The difference with the maximal stable configuration.

OUTPUT:

SandpileConfig

EXAMPLES:

```
sage: S = sandpiles.Cycle(3)
sage: c = SandpileConfig(S, [1,2])
sage: S.max_stable()
{1: 1, 2: 1}
sage: c.dualize()
{1: 0, 2: -1}
sage: S.max_stable() - c == c.dualize()
True
```

equivalent_recurrent (with_firing_vector=False)
The recurrent configuration equivalent to the given configuration. Optionally, return the corresponding firing vector.

INPUT:

- with_firing_vector – (default: False) boolean
OUTPUT:

SandpileConfig or [SandpileConfig, firing_vector]

EXAMPLES:

```sage
sage: S = sandpiles.Diamond()
sage: c = SandpileConfig(S, [0,0,0])
sage: c.equivalent_recurrent() == S.identity()
True
sage: x = c.equivalent_recurrent(True)
sage: r = vector([x[0][v] for v in S.nonsink_vertices()])
sage: f = vector([x[1][v] for v in S.nonsink_vertices()])
sage: cv = vector(c.values())
sage: r == cv - f*S.reduced_laplacian()
True
```

Note: Let $L$ be the reduced Laplacian, $c$ the initial configuration, $r$ the returned configuration, and $f$ the firing vector. Then $r = c - f \cdot L$.

---

**equivalent_superstable** *(with_firing_vector=False)*

The equivalent superstable configuration. Optionally, return the corresponding firing vector.

INPUT:

with_firing_vector= *(default: False)* boolean

OUTPUT:

SandpileConfig or [SandpileConfig, firing_vector]

EXAMPLES:

```sage
sage: S = sandpiles.Diamond()
sage: m = S.max_stable()
sage: m.equivalent_superstable().is_superstable()
True
sage: x = m.equivalent_superstable(True)
sage: s = vector(x[0].values())
sage: f = vector(x[1].values())
sage: mv = vector(m.values())
sage: s == mv - f*S.reduced_laplacian()
True
```

Note: Let $L$ be the reduced Laplacian, $c$ the initial configuration, $s$ the returned configuration, and $f$ the firing vector. Then $s = c - f \cdot L$.

---

**fire_script** *(sigma)*

Fire the given script. In other words, fire each vertex the number of times indicated by $\sigma$.

INPUT:

sigma – SandpileConfig or (list or dict representing a SandpileConfig)

OUTPUT:

SandpileConfig

EXAMPLES:
sage: S = sandpiles.Cycle(4)
sage: c = SandpileConfig(S, [1,2,3])
sage: c.unstable()
[2, 3]
sage: c.fire_script(SandpileConfig(S,[0,1,1]))
{1: 2, 2: 1, 3: 2}
sage: c.fire_script(SandpileConfig(S,[2,0,0])) == c.fire_vertex(1).fire_vertex(1)
True

fire_unstable()
Fire all unstable vertices.

OUTPUT:
SandpileConfig

EXAMPLES:

```
sage: S = sandpiles.Cycle(4)
sage: c = SandpileConfig(S, [1,2,3])
sage: c.fire_unstable()
{1: 2, 2: 1, 3: 2}
```

fire_vertex(v)
Fire the given vertex.

INPUT:

v – vertex

OUTPUT:
SandpileConfig

EXAMPLES:

```
sage: S = sandpiles.Cycle(3)
sage: c = SandpileConfig(S, [1,2])
sage: c.fire_vertex(2)
{1: 2, 2: 0}
```

static help( verbose=True)
List of SandpileConfig methods. If verbose, include short descriptions.

INPUT:

verbose – (default: True) boolean

OUTPUT:
printed string

EXAMPLES:

```
sage: SandpileConfig.help()  
Shortcuts for SandpileConfig operations:
- ~c -- stabilize
c & d -- add and stabilize
c * c -- add and find equivalent recurrent
c^k -- add k times and find equivalent recurrent
(taking inverse if k is negative)
```
For detailed help with any method FOO listed below, enter "SandpileConfig.FOO?" or enter "c.FOO?" for any SandpileConfig c.

- **add_random** -- Add one grain of sand to a random vertex.
- **burst_size** -- The burst size of the configuration with respect to the given vertex.
- **deg** -- The degree of the configuration.
- **dualize** -- The difference with the maximal stable configuration.
- **equivalent_recurrent** -- The recurrent configuration equivalent to the given configuration.
- **equivalent_superstable** -- The equivalent superstable configuration.
- **fire_script** -- Fire the given script.
- **fire_unstable** -- Fire all unstable vertices.
- **fire_vertex** -- Fire the given vertex.
- **help** -- List of SandpileConfig methods.
- **is_recurrent** -- Is the configuration recurrent?
- **is_stable** -- Is the configuration stable?
- **is_superstable** -- Is the configuration superstable?
- **is_symmetric** -- Is the configuration symmetric?
- **order** -- The order of the equivalent recurrent element.
- **sandpile** -- The configuration's underlying sandpile.
- **show** -- Show the configuration.
- **stabilize** -- The stabilized configuration.
- **support** -- The vertices containing sand.
- **unstable** -- The unstable vertices.
- **values** -- The values of the configuration as a list.

### is_recurrent()

Is the configuration recurrent?

**OUTPUT:**

boolean

**EXAMPLES:**

```
sage: S = sandpiles.Diamond()
sage: S.identity().is_recurrent()  # True
sage: S.zero_config().is_recurrent()  # False
```

### is_stable()

Is the configuration stable?

**OUTPUT:**

boolean

**EXAMPLES:**

```
sage: S = sandpiles.Diamond()
sage: S.max_stable().is_stable()  # True
sage: (2*S.max_stable()).is_stable()  # False
```

(continued from previous page)

```python
sage: (S.max_stable() & S.max_stable()).is_stable()
True
```

**is_superstable()**

Is the configuration superstable?

OUTPUT:

boolean

EXAMPLES:

```python
sage: S = sandpiles.Diamond()
sage: S.zero_config().is_superstable()
True
```

**is_symmetric**(orbits)

Is the configuration symmetric? Return True if the values of the configuration are constant over the vertices in each sublist of orbits.

INPUT:

orbits – list of lists of vertices

OUTPUT:

boolean

EXAMPLES:

```python
sage: S = Sandpile({0: {},
....: 1: {0: 1, 2: 1, 3: 1},
....: 2: {1: 1, 3: 1, 4: 1},
....: 3: {1: 1, 2: 1, 4: 1},
....: 4: {2: 1, 3: 1}})
sage: c = SandpileConfig(S, [1, 2, 2, 3])
sage: c.is_symmetric([[2,3]])
True
```

**order()**

The order of the equivalent recurrent element.

OUTPUT:

integer

EXAMPLES:

```python
sage: S = sandpiles.Diamond()
sage: c = SandpileConfig(S,[2,0,1])
sage: c.order()
4
sage: ~(c + c + c + c) == S.identity()
True
sage: c = SandpileConfig(S,[1,1,0])
sage: c.order()
1
c.is_recurrent()  # False
sage: c.equivalent_recurrent() == S.identity()
True
```
sandpile()
The configuration's underlying sandpile.

OUTPUT:
Sandpile

EXAMPLES:

```python
sage: S = sandpiles.Diamond()
sage: c = S.identity()
sage: c.sandpile()
Diamond sandpile graph: 4 vertices, sink = 0
sage: c.sandpile() == S
True
```

show (sink=True, colors=True, heights=False, directed=None, **kwds)
Show the configuration.

INPUT:

- sink – (default: True) whether to show the sink
- colors – (default: True) whether to color-code the amount of sand on each vertex
- heights – (default: False) whether to label each vertex with the amount of sand
- directed – (optional) whether to draw directed edges
- kwds – (optional) arguments passed to the show method for Graph

EXAMPLES:

```python
sage: S = sandpiles.Diamond()
sage: c = S.identity()
sage: c.show()
sage: c.show(directed=False)
sage: c.show(sink=False,colors=False,heights=True)
```

stabilize (with_firing_vector=False)
The stabilized configuration. Optionally returns the corresponding firing vector.

INPUT:

with_firing_vector – (default: False) boolean

OUTPUT:
SandpileConfig or [SandpileConfig, firing_vector]

EXAMPLES:

```python
sage: S = sandpiles.House()
sage: c = 2*S.max_stable()
sage: c._set_stabilize()
_s stabilize' in c.__dict__
True
sage: S = sandpiles.House()
sage: c = S.max_stable() + S.identity()
sage: c.stabilize(True)
[{{1: 1, 2: 2, 3: 2, 4: 1}, {1: 2, 2: 2, 3: 3, 4: 3}}]
sage: S.max_stable() & S.identity() == c.stabilize()
True
```
support()
The vertices containing sand.

OUTPUT:
list - support of the configuration

EXAMPLES:

```
sage: S = sandpiles.Diamond()
sage: c = S.identity()
sage: c
{1: 2, 2: 2, 3: 0}
sage: c.support()
[1, 2]
```

unstable()
The unstable vertices.

OUTPUT:
list of vertices

EXAMPLES:

```
sage: S = sandpiles.Cycle(4)
sage: c = SandpileConfig(S, [1,2,3])
sage: c.unstable()
[2, 3]
```

values()
The values of the configuration as a list.
The list is sorted in the order of the vertices.

OUTPUT:
list of integers

EXAMPLES:

```
sage: S = Sandpile({'a':['c','b'], 'b':['c','a'], 'c':['a']),'a')
sage: c = SandpileConfig(S, {'b':1, 'c':2})
sage: c
{'b': 1, 'c': 2}
sage: c.values()
[1, 2]
sage: S.nonsink_vertices()
['b', 'c']
```

```
class sage.sandpiles.sandpile.SandpileDivisor(S, D)
    Bases: dict

    Class for divisors on a sandpile.
```
**Dcomplex()**

The support-complex. (See NOTE.)

**OUTPUT:**

simplicial complex

**EXAMPLES:**

```
sage: S = sandpiles.House()
sage: p = SandpileDivisor(S, [1,2,1,0,0]).Dcomplex()
sage: p.homology()
{0: 0, 1: \( \mathbb{Z} \times \mathbb{Z} \), 2: 0}
sage: p.f_vector()
[1, 5, 10, 4]
sage: p.betti()
{0: 1, 1: 2, 2: 0}
```

**Note:** The “support-complex” is the simplicial complex determined by the supports of the linearly equivalent effective divisors.

**add_random**(distrib=None)

Add one grain of sand to a random vertex.

**INPUT:**

- distrib – (optional) list of nonnegative numbers representing a probability distribution on the vertices

**OUTPUT:**

SandpileDivisor

**EXAMPLES:**

```
sage: s = sandpiles.Complete(4)
sage: D = s.zero_div()
sage: D.add_random() # random
{0: 0, 1: 0, 2: 1, 3: 0}
sage: D.add_random([0.1,0.1,0.1,0.7]) # random
{0: 0, 1: 0, 2: 0, 3: 1}
```

**Warning:** If `distrib` is not `None`, the user is responsible for assuring the sum of its entries is 1.

**betti()**

The Betti numbers for the support-complex. (See NOTE.)

**OUTPUT:**

dictionary of integers

**EXAMPLES:**

```
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [2,0,1])
sage: D.betti()
{0: 1, 1: 1}
```
Note: The “support-complex” is the simplicial complex determined by the supports of the linearly equivalent effective divisors.

\texttt{deg()}

The degree of the divisor.

OUTPUT:

integer

EXAMPLES:

```
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.deg()
6
```

dualize()

The difference with the maximal stable divisor.

OUTPUT:

SandpileDivisor

EXAMPLES:

```
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.dualize()
{0: 0, 1: -1, 2: -2}
sage: S.max_stable_div() - D == D.dualize()
True
```

effective_div (\texttt{verbose}=True, \texttt{with_firing_vectors}=False)

All linearly equivalent effective divisors. If \texttt{verbose} is \texttt{False}, the divisors are converted to lists of integers. If \texttt{with_firing_vectors} is \texttt{True} then a list of firing vectors is also given, each of which prescribes the vertices to be fired in order to obtain an effective divisor.

INPUT:

- \texttt{verbose} – (default: \texttt{True}) boolean
- \texttt{with_firing_vectors} – (default: \texttt{False}) boolean

OUTPUT:

list (of divisors)

EXAMPLES:

```
sage: s = sandpiles.Complete(4)
sage: D = SandpileDivisor(s,[4,2,0,0])
sage: sorted(D.effective_div(), key=str)
[[{0: 0, 1: 2, 2: 0, 3: 4},
  {0: 0, 1: 2, 2: 4, 3: 0},
  {0: 0, 1: 6, 2: 0, 3: 0},
  {0: 1, 1: 3, 2: 1, 3: 1},
  {0: 2, 1: 0, 2: 2, 3: 2},
  {0: 4, 1: 2, 2: 0, 3: 0}]
sage: sorted(D.effective_div(False))
```

(continues on next page)
fire_script (sigma)

Fire the given script. In other words, fire each vertex the number of times indicated by sigma.

INPUT:

sigma – SandpileDivisor or (list or dict representing a SandpileDivisor)

OUTPUT:

SandpileDivisor

EXAMPLES:

```
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.unstable()
[1, 2]
sage: D.fire_script([0,1,1])
{0: 3, 1: 1, 2: 2}
sage: D.fire_script(SandpileDivisor(S,[2,0,0])) == D.fire_vertex(0).fire_
˓→vertex(0)
True
```

fire_unstable ()

Fire all unstable vertices.

OUTPUT:

SandpileDivisor

EXAMPLES:
**fire_vertex** ($v$)

Fire the given vertex.

**INPUT:**

$v$ – vertex

**OUTPUT:**

SandpileDivisor

**EXAMPLES:**

```plaintext
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.fire_vertex(1)
{0: 2, 1: 0, 2: 4}
```

**static help** *(verbose=True)*

List of SandpileDivisor methods. If verbose, include short descriptions.

**INPUT:**

verbose – (default: True) boolean

**OUTPUT:**

printed string

**EXAMPLES:**

```plaintext
sage: SandpileDivisor.help()
For detailed help with any method FOO listed below, enter "SandpileDivisor.FOO?" or enter "D.FOO?" for any SandpileDivisor D.

Dcomplex -- The support-complex.
add_random -- Add one grain of sand to a random vertex.
betti -- The Betti numbers for the support-complex.
deg -- The degree of the divisor.
dualize -- The difference with the maximal stable divisor.
effective_div -- All linearly equivalent effective divisors.
fire_script -- Fire the given script.
fire_unstable -- Fire all unstable vertices.
fire_vertex -- Fire the given vertex.
help -- List of SandpileDivisor methods.
is_alive -- Is the divisor stabilizable?
is_linearly_equivalent -- Is the given divisor linearly equivalent?
is_q_reduced -- Is the divisor q-reduced?
is_symmetric -- Is the divisor symmetric?
is_weierstrass_pt -- Is the given vertex a Weierstrass point?
polytope -- The polytope determining the complete linear system.
polytope_integer_pts -- The integer points inside divisor's polytope.
q_reduced -- The linearly equivalent q-reduced divisor.
rang -- The rank of the divisor.
sandpile -- The divisor's underlying sandpile.
show -- Show the divisor.
```
is_alive\( (\text{cycle=}False) \)

Is the divisor stabilizable? In other words, will the divisor stabilize under repeated firings of all unstable vertices? Optionally returns the resulting cycle.

**INPUT:**

- cycle – (default: False) boolean

**OUTPUT:**

boolean or optionally, a list of SandpileDivisors

**EXAMPLES:**

```
sage: S = sandpiles.Complete(4)
sage: D = SandpileDivisor(S, [0: 4, 1: 3, 2: 3, 3: 2])
sage: D.is_alive()
True
sage: D.is_alive(True)
[[0: 4, 1: 3, 2: 3, 3: 2], [0: 3, 1: 2, 2: 2, 3: 5], [0: 1, 1: 4, 2: 4, 3: 3]]
```

is_linearly_equivalent\( (D, \text{with\_firing\_vector=}False) \)

Is the given divisor linearly equivalent? Optionally, returns the firing vector. (See NOTE.)

**INPUT:**

- D – SandpileDivisor or list, tuple, etc. representing a divisor
- with_firing_vector – (default: False) boolean

**OUTPUT:**

boolean or integer vector

**EXAMPLES:**

```
sage: s = sandpiles.Complete(3)
sage: D = SandpileDivisor(s,[2,0,0])
sage: D.is_linearly_equivalent([0,1,1])
True
sage: D.is_linearly_equivalent([0,1,1],True)
(1, 0, 0)
sage: v = vector(D.is_linearly_equivalent([0,1,1],True))
sage: vector(D.values()) - s.laplacian()*v
(0, 1, 1)
sage: D.is_linearly_equivalent([0,0,0])
False
sage: D.is_linearly_equivalent([0,0,0],True)
()```
is_q_reduced()

Is the divisor $q$-reduced? This would mean that $self = c + kq$ where $c$ is superstable, $k$ is an integer, and $q$ is the sink vertex.

OUTPUT:

boolean

EXAMPLES:

```python
sage: s = sandpiles.Complete(4)
sage: D = SandpileDivisor(s,[2,-3,2,0])
sage: D.is_q_reduced()
False
sage: SandpileDivisor(s,[10,0,1,2]).is_q_reduced()
True
```

For undirected or, more generally, Eulerian graphs, $q$-reduced divisors are linearly equivalent if and only if they are equal. The same does not hold for general directed graphs:

```python
sage: s = Sandpile({0:{1:1,2:1}},{1:{0:1,3:1},{2:{0:1,3:1,4:1}}})
sage: D = SandpileDivisor(s,[-1,1])
sage: Z = s.zero_div()
sage: D.is_q_reduced()
True
sage: Z.is_q_reduced()
True
sage: D == Z
False
sage: D.is_linearly_equivalent(Z)
True
```

is_symmetric(orbits)

Is the divisor symmetric? Return True if the values of the configuration are constant over the vertices in each sublist of orbits.

INPUT:

orbits – list of lists of vertices

OUTPUT:

boolean

EXAMPLES:

```python
sage: S = sandpiles.House()
sage: S.dict()
{0: {1: 1, 2: 1},
  1: {0: 1, 3: 1},
  2: {0: 1, 3: 1, 4: 1},
```
is\_weierstrass\_pt(v='sink')
Is the given vertex a Weierstrass point?

INPUT:

v – (default: sink) vertex

OUTPUT:

boolean

EXAMPLES:

\begin{verbatim}
sage: s = sandpiles.House()
sage: K = s.canonical_divisor()
sage: K.weierstrass_rank_seq()   # sequence at the sink vertex, 0
(1, 0, -1)
sage: K.is_weierstrass_pt()
False
sage: K.weierstrass_rank_seq(4)
(1, 0, 0, -1)
sage: K.is_weierstrass_pt(4)
True
\end{verbatim}

Note: The vertex \( v \) is a (generalized) Weierstrass point for divisor \( D \) if the sequence of ranks \( r(D - n v) \) for \( n = 0, 1, 2, \ldots \) is not \( r(D), r(D) - 1, \ldots, 0, -1, -1, \ldots \).

polytope()

The polytope determining the complete linear system.

OUTPUT:

polytope

EXAMPLES:

\begin{verbatim}
sage: s = sandpiles.Complete(4)
sage: D = SandpileDivisor(s,[4,2,0,0])
sage: p = D.polytope()
sage: p.inequalities()
(An inequality (-3, 1, 1) x + 2 >= 0,  
 An inequality (1, 1, 1) x + 4 >= 0,  
 An inequality (1, -3, 1) x + 0 >= 0,  
 An inequality (1, 1, -3) x + 0 >= 0) 
sage: D = SandpileDivisor(s,[-1,0,0,0])
sage: D.polytope()
The empty polyhedron in QQ^3
\end{verbatim}

Note: For a divisor \( D \), this is the intersection of (i) the polyhedron determined by the system of inequalities \( L^t x \leq D \) where \( L^t \) is the transpose of the Laplacian with (ii) the hyperplane \( x_{\text{sink vertex}} = 0 \). The
polytope is thought of as sitting in \((n-1)\)-dimensional Euclidean space where \(n\) is the number of vertices.

**polytope_integer_pts()**

The integer points inside divisor's polytope. The polytope referred to here is the one determining the divisor's complete linear system (see the documentation for polytope).

OUTPUT:

tuple of integer vectors

EXAMPLES:

```
sage: s = sandpiles.Complete(4)
sage: D = SandpileDivisor(s,[4,2,0,0])
sage: sorted(D.polytope_integer_pts())
[(-2, -1, -1),
(-1, -2, -1),
(-1, -1, -2),
(-1, -1, -1),
(0, -1, -1),
(0, 0, 0)]
sage: D = SandpileDivisor(s,[-1,0,0,0])
sage: D.polytope_integer_pts()
()
```

**q_reduced** *(verbose=True)*

The linearly equivalent \(q\)-reduced divisor.

INPUT:

verbose – (default: True) boolean

OUTPUT:

SandpileDivisor or list representing SandpileDivisor

EXAMPLES:

```
sage: s = sandpiles.Complete(4)
sage: D = SandpileDivisor(s,[2,-3,2,0])
sage: D.q_reduced()
{0: -2, 1: 1, 2: 2, 3: 0}
sage: D.q_reduced(False)
[-2, 1, 2, 0]
```

**Note:** The divisor \(D\) is \(q\)reduced if \(D = c + kq\) where \(c\) is superstable, \(k\) is an integer, and \(q\) is the sink.

**rank** *(with_witness=False)*

The rank of the divisor. Optionally returns an effective divisor \(E\) such that \(D - E\) is not winnable (has an empty complete linear system).

INPUT:

with_witness – (default: False) boolean

OUTPUT:

integer or (integer, SandpileDivisor)

EXAMPLES:
```python
sage: S = sandpiles.Complete(4)
sage: D = SandpileDivisor(S,[4,2,0,0])
sage: D.rank()
3
sage: D.rank(True)
(3, {0: 3, 1: 0, 2: 1, 3: 0})
sage: E = _[1]
sage: (D - E).rank()
-1
```

Riemann-Roch theorem::

```python
sage: D.rank() - (S.canonical_divisor()-D).rank() == D.deg() + 1 - S.genus()
True
```

Riemann-Roch theorem::

```python
sage: D.rank() - (S.canonical_divisor()-D).rank() == D.deg() + 1 - S.genus()
True
sage: S = Sandpile({0:[1,1,1,2],1:[0,0,0,1,1,1,2,2],2:[2,2,1,1,0]},0) # multigraph with loops
sage: D = SandpileDivisor(S,[4,2,0])
sage: D.rank(True)
(2, {0: 1, 1: 1, 2: 1})
sage: S = Sandpile({0:[1,2], 1:[0,2,2], 2: [0,1],0} # directed graph
sage: S.is_undirected()
False
sage: D = SandpileDivisor(S,[0,2,0])
sage: D.effective_div()
[(0: 0, 1: 2, 2: 0), (0: 2, 1: 0, 2: 0)]
```

**Note:** The rank of a divisor $D$ is -1 if $D$ is not linearly equivalent to an effective divisor (i.e., the dollar game represented by $D$ is unwinnable). Otherwise, the rank of $D$ is the largest integer $r$ such that $D - E$ is linearly equivalent to an effective divisor for all effective divisors $E$ with $\deg(E) = r$.

**sandpile()**

The divisor’s underlying sandpile.

**OUTPUT:**

Sandpile

**EXAMPLES:**

```python
sage: S = sandpiles.Diamond()
sage: D = SandpileDivisor(S,[1,-2,0,3])
sage: D.sandpile()
Diamond sandpile graph: 4 vertices, sink = 0
sage: D.sandpile() == S
True
```
show (heights=True, directed=None, **kwds)
Show the divisor.

INPUT:

• heights – (default: True) whether to label each vertex with the amount of sand
• directed – (optional) whether to draw directed edges
• kwds – (optional) arguments passed to the show method for Graph

EXAMPLES:

```
sage: S = sandpiles.Diamond()
sage: D = SandpileDivisor(S,[1,-2,0,2])
sage: D.show(graph_border=True,vertex_size=700,directed=False)
```

simulate_threshold (distrib=None)
The first unstabilizable divisor in the closed Markov chain. (See NOTE.)

INPUT:

distrib – (optional) list of nonnegative numbers representing a probability distribution on the vertices

OUTPUT:
SandpileDivisor

EXAMPLES:

```
sage: s = sandpiles.Complete(4)
sage: D = s.zero_div()
sage: D.simulate_threshold() # random
{0: 2, 1: 3, 2: 1, 3: 2}
sage: n(mean([D.simulate_threshold().deg() for _ in range(10)])) # random
7.10000000000000
sage: n(s.stationary_density()*s.num_verts())
6.93750000000000
```

Note: Starting at self, repeatedly choose a vertex and add a grain of sand to it. Return the first unstabilizable divisor that is reached. Also see the markov_chain method for the underlying sandpile.

stabilize (with_firing_vector=False)
The stabilization of the divisor. If not stabilizable, return an error.

INPUT:

with_firing_vector – (default: False) boolean

EXAMPLES:

```
sage: s = sandpiles.Complete(4)
sage: D = SandpileDivisor(s,[0,3,0,0])
sage: D.stabilize()
{0: 1, 1: 0, 2: 1, 3: 1}
sage: D.stabilize(with_firing_vector=True)
[({0: 1, 1: 0, 2: 1, 3: 1}, {0: 0, 1: 1, 2: 0, 3: 0})]
```

support ()
List of vertices at which the divisor is nonzero.
OUTPUT:
list representing the support of the divisor

EXAMPLES:
```python
sage: S = sandpiles.Cycle(4)
sage: D = SandpileDivisor(S, [0,0,1,1])
sage: D.support()
[2, 3]
sage: S.vertices()
[0, 1, 2, 3]
```

`unstable()`
The unstable vertices.

OUTPUT:
list of vertices

EXAMPLES:
```python
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.unstable()
[1, 2]
```

`values()`
The values of the divisor as a list.
The list is sorted in the order of the vertices.

OUTPUT:
list of integers
boolean

EXAMPLES:
```python
sage: S = Sandpile({'a':['c','b'], 'b':['c','a'], 'c':['a']},'a')
sage: D = SandpileDivisor(S, {'a':0, 'b':1, 'c':2})
sage: D
{'a': 0, 'b': 1, 'c': 2}
sage: D.values()
[0, 1, 2]
sage: S.vertices()
['a', 'b', 'c']
```

`weierstrass_div` *(verbose=True)*
The Weierstrass divisor. Its value at a vertex is the weight of that vertex as a Weierstrass point. (See SandpileDivisor.weierstrass_gap_seq.)

INPUT:
verbose – (default: True) boolean

OUTPUT:
SandpileDivisor

EXAMPLES:
sage: s = sandpiles.Diamond()
sage: D = SandpileDivisor(s,[4,2,1,0])
sage: [D.weierstrass_rank_seq(v) for v in s]
[(5, 4, 3, 2, 1, 0, 0, -1),
 (5, 4, 3, 2, 1, 0, -1),
 (5, 4, 3, 2, 1, 0, 0, -1),
 (5, 4, 3, 2, 1, 0, 0, -1)]
sage: D.weierstrass_div()
{0: 1, 1: 0, 2: 2, 3: 1}
sage: k5 = sandpiles.Complete(5)
sage: K = k5.canonical_divisor()
sage: K.weierstrass_div()
{0: 9, 1: 9, 2: 9, 3: 9, 4: 9}

\texttt{weierstrass\_gap\_seq}(v='\texttt{sink}',\ weight=True)

The Weierstrass gap sequence at the given vertex. If \texttt{weight} is \texttt{True}, then also compute the weight of each gap value.

**INPUT:**

- \texttt{v} – (default: \texttt{sink}) vertex
- \texttt{weight} – (default: \texttt{True}) boolean

**OUTPUT:**

list or (list of list) of integers

**EXAMPLES:**

sage: s = sandpiles.Cycle(4)
sage: D = SandpileDivisor(s,[2,0,0,0])
sage: [D.weierstrass_gap_seq(v,False) for v in s.vertices()]
[(1, 3), (1, 2), (1, 3), (1, 2)]
sage: [D.weierstrass_gap_seq(v) for v in s.vertices()]
[((1, 3), 1), ((1, 2), 2), ((1, 3), 1), ((1, 2), 0)]
sage: D.weierstrass_gap_seq() # gap sequence at sink vertex, 0
((1, 3), 1)
sage: D.weierstrass_rank_seq() # rank sequence at the sink vertex
(1, 0, 0, -1)

\textbf{Note:} The integer \(k\) is a Weierstrass gap for the divisor \(D\) at vertex \(v\) if the rank of \(D - (k - 1)v\) does not equal the rank of \(D - kv\). Let \(r\) be the rank of \(D\) and let \(k_i\) be the \(i\)-th gap at \(v\). The Weierstrass weight of \(v\) for \(D\) is the sum of \((k_i - i)\) as \(i\) ranges from 1 to \(r + 1\). It measure the difference between the sequence \(r, r - 1, ..., 0, -1, -1, ...\) and the rank sequence \(\text{rank}(D), \text{rank}(D - v), \text{rank}(D - 2v), \ldots\)

\texttt{weierstrass\_pts}(\textit{with\_rank\_seq}=False)

The Weierstrass points (vertices). Optionally, return the corresponding rank sequences.

**INPUT:**

\texttt{with\_rank\_seq} – (default: \texttt{False}) boolean

**OUTPUT:**

tuple of vertices or list of (vertex, rank sequence)

**EXAMPLES:**
 sage: s = sandpiles.House()
sage: K = s.canonical_divisor()
sage: K.weierstrass_pts()
(4,
)
sage: K.weierstrass_pts(True)
[(4, (1, 0, 0, -1))]

Note: The vertex \(v\) is a (generalized) Weierstrass point for divisor \(D\) if the sequence of ranks \(r(D - nv)\) for \(n = 0, 1, 2, \ldots\) is not \(r(D), r(D) - 1, \ldots, 0, -1, -1, \ldots\)

**weierstrass_rank_seq** \((v='sink')\)
The Weierstrass rank sequence at the given vertex. Computes the rank of the divisor \(D - nv\) starting with \(n = 0\) and ending when the rank is \(-1\).

**INPUT:**

\(v\) – (default: sink) vertex

**OUTPUT:**
tuple of int

**EXAMPLES:**

 sage: s = sandpiles.House()
sage: K = s.canonical_divisor()
sage: [K.weierstrass_rank_seq(v) for v in s.vertices()]
[(1, 0, -1), (1, 0, -1), (1, 0, -1), (1, 0, -1), (1, 0, 0, -1)]

sage.sandpiles.sandpile.admissible_partitions\((S, k)\)
The partitions of the vertices of \(S\) into \(k\) parts, each of which is connected.

**INPUT:**

\(S\) – Sandpile
\(k\) – integer

**OUTPUT:**
list of partitions

**EXAMPLES:**

 sage: from sage.sandpiles.sandpile import admissible_partitions
 sage: S = sandpiles.Cycle(4)
sage: P = [admissible_partitions(S, i) for i in [2, 3, 4]]
sage: P

(continues on next page)
sage.sandpiles.sandpile.aztec_sandpile(n)

The aztec diamond graph.

INPUT:

n – integer

OUTPUT:

dictionary for the aztec diamond graph

EXAMPLES:

```python
sage: from sage.sandpiles.sandpile import aztec_sandpile
global T
sage: T = aztec_sandpile(2)
sage: sorted(len(v) for u, v in T.items())

[3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 8]
sage: Sandpile(T,(0, 0)).group_order()

4542720
```

Note: This is the aztec diamond graph with a sink vertex added. Boundary vertices have edges to the sink so that each vertex has degree 4.

sage.sandpiles.sandpile.firing_graph(S, eff)

Creates a digraph with divisors as vertices and edges between two divisors \( D \) and \( E \) if firing a single vertex in \( D \) gives \( E \).

INPUT:

S – Sandpile

eff – list of divisors

OUTPUT:

DiGraph

EXAMPLES:

```python
sage: S = sandpiles.Cycle(6)
sage: D = SandpileDivisor(S, [1,1,1,1,2,0])
sage: eff = D.effective_div()
sage: firing_graph(S,eff).show3d(edge_size=.005,vertex_size=0.01)  # long time
```
sage.sandpiles.sandpile.glue_graphs\( (g, h, glue_g, glue_h) \)
Glue two graphs together.

**INPUT:**
- \( g, h \) – dictionaries for directed multigraphs
- \( glue_h, glue_g \) – dictionaries for a vertex

**OUTPUT:**
dictionary for a directed multigraph

**EXAMPLES:**

```python
sage: from sage.sandpiles.sandpile import glue_graphs
sage: x = {0: {}, 1: {0: 1}, 2: {0: 1, 1: 1}, 3: {0: 1, 1: 1, 2: 1}}

sage: y = {0: {}, 1: {0: 2}, 2: {1: 2}, 3: {0: 1, 2: 1}}

sage: glue_x = {1: 1, 3: 2}

sage: glue_y = {0: 1, 1: 2, 3: 1}

sage: z = glue_graphs(x, y, glue_x, glue_y); z

{'sink': {}, 'x0': {'sink': 1, 'x1': 1, 'x3': 2, 'y1': 2, 'y3': 1},
 'x1': {'x0': 1}, 'x2': {'x0': 1, 'x1': 1},
 'x3': {'x0': 1, 'x1': 1, 'x2': 1},
 'y1': {'sink': 2}, 'y2': {'y1': 2},
 'y3': {'sink': 1, 'y2': 1}}

sage: S = Sandpile(z, 'sink')

sage: S.h_vector()

[1, 6, 17, 31, 41, 41, 31, 17, 6, 1]

sage: S.resolution()

'R^1 <-- R^7 <-- R^21 <-- R^35 <-- R^35 <-- R^21 <-- R^7 <-- R^1'
```

**Note:** This method makes a dictionary for a graph by combining those for \( g \) and \( h \). The sink of \( g \) is replaced by a vertex that is connected to the vertices of \( g \) as specified by \( glue_g \) the vertices of \( h \) as specified in \( glue_h \). The sink of the glued graph is \('sink'\).

Both \( glue_g \) and \( glue_h \) are dictionaries with entries of the form \( v:w \) where \( v \) is the vertex to be connected to and \( w \) is the weight of the connecting edge.

sage.sandpiles.sandpile.min_cycles\( (G, v) \)
Minimal length cycles in the digraph \( G \) starting at vertex \( v \).

**INPUT:**
- \( G \) – DiGraph
- \( v \) – vertex of \( G \)

**OUTPUT:**
list of lists of vertices

**EXAMPLES:**

```python
sage: from sage.sandpiles.sandpile import min_cycles, sandlib

sage: T = sandlib('gor')

sage: [min_cycles(T, i) for i in T.vertices()]

[[], [[1, 3]], [[2, 3, 1], [2, 3]], [[3, 1], [3, 2]]]
```
sage.sandpiles.sandpile.parallel_firing_graph(S, eff)

Creates a digraph with divisors as vertices and edges between two divisors \( D \) and \( E \) if firing all unstable vertices in \( D \) gives \( E \).

INPUT:

\( S \) – Sandpile

\( \text{eff} \) – list of divisors

OUTPUT:

DiGraph

EXAMPLES:

```
sage: S = sandpiles.Cycle(6)
sage: D = SandpileDivisor(S, [1,1,1,1,2,0])
sage: eff = D.effective_div()
sage: parallel_firing_graph(S,eff).show3d(edge_size=.005,vertex_size=0.01)  # long
˓→time```

sage.sandpiles.sandpile.partition_sandpile(S, p)

Each set of vertices in \( p \) is regarded as a single vertex, with an edge between \( A \) and \( B \) if some element of \( A \) is connected by an edge to some element of \( B \) in \( S \).

INPUT:

\( S \) – Sandpile

\( p \) – partition of the vertices of \( S \)

OUTPUT:

Sandpile

EXAMPLES:

```
sage: from sage.sandpiles.sandpile import admissible_partitions, partition_
˓→sandpile
sage: S = sandpiles.Cycle(4)
sage: P = [admissible_partitions(S, i) for i in [2,3,4]]
sage: for p in P:
       ....: sum([partition_sandpile(S, i).betti(verbose=False)[-1] for i in p])
       6
       8
       3
sage: S.betti()
0 1 2 3
-------------------------------
0: 1  -  -  - 
1:  -  6  8  3
-------------------------------
total: 1  6  8  3```

sage.sandpiles.sandpile.random_DAG(num_verts, p=0.5, weight_max=1)

A random directed acyclic graph with \( \text{num_verts} \) vertices. The method starts with the sink vertex and adds vertices one at a time. Each vertex is connected only to only previously defined vertices, and the probability of each possible connection is given by the argument \( p \). The weight of an edge is a random integer between 1 and \( \text{weight_max} \).

INPUT:
• `num_verts` – positive integer
• `p` – (default: 0.5) real number such that $0 < p \leq 1$
• `weight_max` – (default: 1) positive integer

OUTPUT:
a dictionary, encoding the edges of a directed acyclic graph with sink 0

EXAMPLES:

```sage
d = DiGraph(random_DAG(5, .5)); d
Digraph on 5 vertices
```

```python
sage.sandpiles.sandpile.sandlib(selector=None)

Returns the sandpile identified by `selector`. If no argument is given, a description of the sandpiles in the sandlib is printed.

INPUT:

`selector` – (optional) identifier or None

OUTPUT:

sandpile or description

EXAMPLES:

```python
sage: from sage.sandpiles.sandpile import sandlib
sage: sandlib()
Sandpiles in the sandlib:
  ci1 : complete intersection, non-DAG but equivalent to a DAG
  generic : generic digraph with 6 vertices
  genus2 : Undirected graph of genus 2
  gor : Gorenstein but not a complete intersection
  kite : generic undirected graphs with 5 vertices
  riemann-roch1 : directed graph with postulation 9 and 3 maximal weight
  riemann-roch2 : directed graph with a superstable not majorized by a maximal
  superstables
  superstable
sage: S = sandlib('gor')
sage: S.resolution()
'R^1 <-- R^5 <-- R^5 <-- R^1'
```

```python
sage.sandpiles.sandpile.triangle_sandpile(n)

A triangular sandpile. Each nonsink vertex has out-degree six. The vertices on the boundary of the triangle are connected to the sink.

INPUT:

`n` – integer

OUTPUT:

Sandpile

EXAMPLES:

```python
sage: from sage.sandpiles.sandpile import triangle_sandpile
sage: T = triangle_sandpile(5)
sage: T.group_order()
135418115000
```
sage.sandpiles.sandpile.wilmes_algorithm(M)

Computes an integer matrix $L$ with the same integer row span as $M$ and such that $L$ is the reduced Laplacian of a directed multigraph.

INPUT:

$M$ – square integer matrix of full rank

OUTPUT:

integer matrix ($L$)

EXAMPLES:

```
sage: P = matrix([[2,3,-7,-3],[5,2,-5,5],[8,2,5,4],[-5,-9,6,6]])
sage: wilmes_algorithm(P)
[ 1642  -13 -1627   -1]
[   -1  1980 -1582  -397]
[   0   -1  1650 -1649]
[   0    0 -1658  1658]
```

REFERENCES:

• [PPW2013]

See also:

• sage.combinat.e_one_star
• sage.combinat.constellation
6.1 Generic dynamical systems on schemes

This is the generic class for dynamical systems and contains the exported constructor functions. The constructor functions can take either polynomials (or rational functions in the affine case) or morphisms from which to construct a dynamical system. If the domain is not specified, it is constructed. However, if you plan on working with points or subvarieties in the domain, it is recommended to specify the domain. For products of projective spaces the domain must be specified.

The initialization checks are always performed by the constructor functions. It is possible, but not recommended, to skip these checks by calling the class initialization directly.

AUTHORS:

- Ben Hutz (July 2017): initial version

```python
class sage.dynamics.arithmetic_dynamics.generic_ds.DynamicalSystem(polys_or_rat_fncts, domain)
```

Bases: `sage.schemes.generic.morphism.SchemeMorphism_polynomial`

Base class for dynamical systems of schemes.

INPUT:

- `polys_or_rat_fncts` – a list of polynomials or rational functions, all of which should have the same parent
- `domain` – an affine or projective scheme, or product of projective schemes, on which `polys` defines an endomorphism. Subschemes are also ok
- `names` – (default: ('X', 'Y')) tuple of strings to be used as coordinate names for a projective space that is constructed

The following combinations of `morphism_or_polys` and `domain` are meaningful:

- `morphism_or_polys` is a `SchemeMorphism`; `domain` is ignored in this case
- `morphism_or_polys` is a list of homogeneous polynomials that define a rational endomorphism of `domain`
- `morphism_or_polys` is a list of homogeneous polynomials and `domain` is unspecified; `domain` is then taken to be the projective space of appropriate dimension over the common parent of the elements in `morphism_or_polys`
- `morphism_or_polys` is a single polynomial or rational function; `domain` is ignored and taken to be a 1-dimensional projective space over the base ring of `morphism_or_polys` with coordinate names given by `names`

EXAMPLES:
\begin{verbatim}
sage: A.<x> = AffineSpace(QQ,1)
sage: f = DynamicalSystem_affine([x^2+1])
sage: type(f)
<class 'sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine_field'>
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2+y^2, y^2])
sage: type(f)
<class 'sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field'>

DynamicalSystem defaults to projective:

\begin{verbatim}
sage: R.<x,y,z> = QQ[]
sage: DynamicalSystem([x^2, y^2, z^2])
Dynamical System of Projective Space of dimension 2 over Rational Field
   Defn: Defined on coordinates by sending (x : y : z) to
         (x^2 : y^2 : z^2)
\end{verbatim}

\begin{verbatim}
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: DynamicalSystem([y, x], domain=A)
Dynamical System of Affine Space of dimension 2 over Rational Field
   Defn: Defined on coordinates by sending (x, y) to
         (y, x)
sage: H = End(A)
sage: DynamicalSystem(H([y, x]))
Dynamical System of Affine Space of dimension 2 over Rational Field
   Defn: Defined on coordinates by sending (x, y) to
         (y, x)
\end{verbatim}

Note that domain is ignored if an endomorphism is passed in:

\begin{verbatim}
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: P2.<x,y> = ProjectiveSpace(CC, 1)
sage: H = End(P2)
sage: f = H([CC.0*x^2, y^2])
sage: g = DynamicalSystem(f, domain=P)
sage: g.domain()
Projective Space of dimension 1 over Complex Field with 53 bits of precision
\end{verbatim}

Constructing a common parent:

\begin{verbatim}
sage: P.<x,y> = ProjectiveSpace(ZZ, 1)
sage: DynamicalSystem([CC.0*x^2, 4/5*y^2])
Dynamical System of Projective Space of dimension 1 over Complex Field with 53 bits of precision
   Defn: Defined on coordinates by sending (x : y) to
         (1.00000000000000*I*x^2 : 0.800000000000000*y^2)
\end{verbatim}
\end{verbatim}

(continues on next page)

P.<x,y> = ProjectiveSpace(GF(5), 1)
P.<t> = GF(25)
DynamicalSystem([GF(5)(3)*x^2, K(t)*y^2])

Dynamical System of Projective Space of dimension 1 over Finite Field in t of size 5^2
Defn: Defined on coordinates by sending (x : y) to (-2*x^2 : (t)*y^2)

as_scheme_morphism()
Return this dynamical system as SchemeMorphism_polynomial.

OUTPUT: SchemeMorphism_polynomial

EXAMPLES:

sage: P.<x,y,z> = ProjectiveSpace(ZZ, 2)
sage: f = DynamicalSystem_projective([x^2, y^2, z^2])
sage: type(f.as_scheme_morphism())
<class 'sage.schemes.projective.projective_morphism.SchemeMorphism_polynomial_projective_space'>

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2-y^2, y^2])
sage: type(f.as_scheme_morphism())
<class 'sage.schemes.projective.projective_morphism.SchemeMorphism_polynomial_projective_space_field'>

sage: P.<x,y> = ProjectiveSpace(GF(5), 1)
sage: f = DynamicalSystem_projective([x^2, y^2])
sage: type(f.as_scheme_morphism())
<class 'sage.schemes.projective.projective_morphism.SchemeMorphism_polynomial_projective_space_finite_field'>

sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: f = DynamicalSystem_affine([x^2-2, y^2])
sage: type(f.as_scheme_morphism())
<class 'sage.schemes.affine.affine_morphism.SchemeMorphism_polynomial_affine_space'>

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: f = DynamicalSystem_affine([x^2-2, y^2])
sage: type(f.as_scheme_morphism())
<class 'sage.schemes.affine.affine_morphism.SchemeMorphism_polynomial_affine_space_field'>

sage: A.<x,y> = AffineSpace(GF(3), 2)
sage: f = DynamicalSystem_affine([x^2-2, y^2])
sage: type(f.as_scheme_morphism())
<class 'sage.schemes.affine.affine_morphism.SchemeMorphism_polynomial_affine_space_finite_field'>

change_ring (R, check=True)
Return a new dynamical system which is this map coerced to R.
If check is True, then the initialization checks are performed.

INPUT:
• \( R \) – ring or morphism

OUTPUT:

A new \texttt{DynamicalSystem\_projective} that is this map coerced to \( R \).

EXAMPLES:

\begin{verbatim}
sage: P.<x,y> = ProjectiveSpace(ZZ, 1)
sage: f = DynamicalSystem_projective([3*x^2, y^2])
sage: f.change_ring(GF(5))
Dynamical System of Projective Space of dimension 1 over Finite Field of size 5
Defn: Defined on coordinates by sending (x : y) to
    (-2*x^2 : y^2)
\end{verbatim}

\texttt{field\_of\_definition\_critical} \texttt{(return\_embedding=False, simplify\_all=False, names='a')}

Return smallest extension of the base field which contains the critical points

Ambient space of dynamical system must be either the affine line or projective line over a number field or finite field.

INPUT:

• \texttt{return\_embedding} – (default: False) boolean; If True, return an embedding of base field of dynamical system into the returned number field or finite field. Note that computing this embedding might be expensive.

• \texttt{simplify\_all} – (default: False) boolean; If True, simplify intermediate fields and also the resulting number field. Note that this is not implemented for finite fields and has no effect.

• \texttt{names} – (optional) string to be used as generator for returned number field or finite field

OUTPUT:

If \texttt{return\_embedding} is False, the field of definition as an absolute number field or finite field. If \texttt{return\_embedding} is True, a tuple \((K, \phi)\) where \(\phi\) is an embedding of the base field in \(K\).

EXAMPLES:

Note that the number of critical points is 2d-2, but (1:0) has multiplicity 2 in this case:

\begin{verbatim}
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem([1/3*x^3 + x*y^2, y^3], domain=P)
sage: f.critical_points()
[(1 : 0)]
sage: N.<a> = f.field_of_definition_critical(); N
Number Field in a with defining polynomial x^2 + 1
sage: g = f.change_ring(N)
sage: g.critical_points()
[(-a : 1), (a : 1), (1 : 0)]
sage: A.<z> = AffineSpace(QQ, 1)
sage: f = DynamicalSystem([z^4 + 2*z^2 + 2], domain=A)
sage: K.<a> = f.field_of_definition_critical(); K
Number Field in a with defining polynomial z^2 + 1
sage: G.<a> = GF(9)
sage: R.<z> = G[]
sage: R.irreducible_element(3, algorithm='first_lexicographic')
z^3 + (a + 1)*z + a
\end{verbatim}

(continues on next page)
sage: A.<x> = AffineSpace(G,1)
sage: f = DynamicalSystem([x^4 + (2*a+2)*x^2 + a*x], domain=A)
sage: f[0].derivative(x).univariate_polynomial().is_irreducible()
True
sage: f.field_of_definition_critical(return_embedding=True, names='b')
(Finite Field in b of size 3^6, Ring morphism:
   From: Finite Field in a of size 3^2
   To:   Finite Field in b of size 3^6
   Defn: a |--> 2*b^5 + 2*b^3 + b^2 + 2*b + 2)

field_of_definition_periodic(n, formal=False, return_embedding=False, simplify_all=False, names='a')

Return smallest extension of the base field which contains all fixed points of the n-th iterate

Ambient space of dynamical system must be either the affine line or projective line over a number field or finite field.

INPUT:

- n -- a positive integer
- formal -- (default: False) boolean; True signals to return number field or finite field over which the formal periodic points are defined, where a formal periodic point is a root of the n-th dynatomic polynomial. False specifies to find number field or finite field over which all periodic points of the n-th iterate are defined
- return_embedding -- (default: False) boolean; If True, return an embedding of base field of dynamical system into the returned number field or finite field. Note that computing this embedding might be expensive.
- simplify_all -- (default: False) boolean; If True, simplify intermediate fields and also the resulting number field. Note that this is not implemented for finite fields and has no effect
- names -- (optional) string to be used as generator for returned number field or finite field

OUTPUT:

If return_embedding is False, the field of definition as an absolute number field or finite field. If return_embedding is True, a tuple (K, phi) where phi is an embedding of the base field in K.

EXAMPLES:

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem([x^2, y^2], domain=P)
sage: f.periodic_points(3, minimal=False)
[(0 : 1), (1 : 0), (1 : 1)]
sage: N.<a> = f.field_of_definition_periodic(3); N
Number Field in a with defining polynomial x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
sage: f.periodic_points(3, minimal=False, R=N)
[(0 : 1),
 (a : 1),
 (a^5 : 1),
 (a^2 : 1),
 (-a^5 - a^4 - a^3 - a^2 - a - 1 : 1),
 (a^4 : 1),
 (1 : 0),
 (a^3 : 1),
 (1 : 1)]
sage: A.<z> = AffineSpace(QQ, 1)
sage: f = DynamicalSystem([(z^2 + 1)/(2*z + 1)], domain=A)
sage: K.<a> = f.field_of_definition_periodic(2); K
Number Field in a with defining polynomial z^4 + 12*z^3 + 39*z^2 + 18*z + 171
sage: F.<b> = f.field_of_definition_periodic(2, formal=True); F
Number Field in b with defining polynomial z^2 + 3*z + 6

sage: G.<a> = GF(4)
sage: A.<x> = AffineSpace(G, 1)
sage: f = DynamicalSystem([x^2 + (a+1)*x + 1], domain=A)
sage: g = f.nth_iterate_map(2)[0]
sage: (g-x).univariate_polynomial().factor()
(x + 1) * (x + a + 1) * (x^2 + a*x + 1)
sage: f.field_of_definition_periodic(2, return_embedding=True, names='b')
(Finite Field in b of size 2^4, Ring morphism:
  From: Finite Field in a of size 2^2
  To:   Finite Field in b of size 2^4
  Defn: a |--> b^2 + b)

field_of_definition_preimage(point, n, return_embedding=False, simplify_all=False, names='a')

Return smallest extension of the base field which contains the n-th preimages of point

Ambient space of dynamical system must be either the affine line or projective line over a number field or finite field.

INPUT:

• point – a point in this map’s domain

• n – a positive integer

• return_embedding – (default: False) boolean; If True, return an embedding of base field of dynamical system into the returned number field or finite field. Note that computing this embedding might be expensive.

• simplify_all – (default: False) boolean; If True, simplify intermediate fields and also the resulting number field. Note that this is not implemented for finite fields and has no effect

• names – (optional) string to be used as generator for returned number field or finite field

OUTPUT:

If return_embedding is False, the field of definition as an absolute number field or finite field. If return_embedding is True, a tuple (K, phi) where phi is an embedding of the base field in K.

EXAMPLES:

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem([[1/3*x^2 + 2/3*x*y, x^2 - 2*y^2], domain=P)
sage: N.<a> = f.field_of_definition_preimage(P(1,1), 2, simplify_all=True); N
Number Field in a with defining polynomial x^8 - 4*x^7 - 128*x^6 + 398*x^5 + 3913*x^4 - 8494*x^3 - 26250*x^2 + 30564*x - 2916

sage: A.<z> = AffineSpace(QQ, 1)
sage: f = DynamicalSystem([z^2], domain=A)
sage: K.<a> = f.field_of_definition_preimage(A(1), 3); K
Number Field in a with defining polynomial z^4 + 1
sage: G = GF(5)
sage: P.<x,y> = ProjectiveSpace(G, 1)
sage: f = DynamicalSystem([x^2 + y^2, y^2], domain=P)
sage: f.field_of_definition_preimage(P(2,1), 2, return_embedding=True, names='a')

(Finite Field in a of size 5^2, Ring morphism:
    From: Finite Field of size 5
    To: Finite Field in a of size 5^2
    Defn: 1 |---> 1)

specialization(D=None, phi=None, homset=None)

Specialization of this dynamical system.

Given a family of maps defined over a polynomial ring. A specialization is a particular member of that family. The specialization can be specified either by a dictionary or a SpecializationMorphism.

INPUT:

• D – (optional) dictionary
• phi – (optional) SpecializationMorphism
• homset – (optional) homset of specialized map

OUTPUT: DynamicalSystem

EXAMPLES:

sage: R.<c> = PolynomialRing(QQ)
sage: P.<x,y> = ProjectiveSpace(R, 1)
sage: f = DynamicalSystem_projective([x^2 + c*y^2,y^2], domain=P)
sage: f.specialization({c:1})
Dynamical System of Projective Space of dimension 1 over Rational Field
Defn: Defined on coordinates by sending (x : y) to
    (x^2 + y^2 : y^2)

6.2 Dynamical systems on affine schemes

An endomorphism of an affine scheme or subscheme determined by polynomials or rational functions.

The main constructor function is given by DynamicalSystem_affine. The constructor function can take polynomials, rational functions, or morphisms from which to construct a dynamical system. If the domain is not specified, it is constructed. However, if you plan on working with points or subvarieties in the domain, it recommended to specify the domain.

The initialization checks are always performed by the constructor functions. It is possible, but not recommended, to skip these checks by calling the class initialization directly.

AUTHORS:

• David Kohel, William Stein
• Volker Braun (2011-08-08): Renamed classes, more documentation, misc cleanups.
• Ben Hutz (2017) relocate code and create new class

class sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine(polys_or_rat_fncts, do-
main)
    Bases: sage.schemes.affine.affine_morphism.SchemeMorphism_polynomial_affine_space.
An endomorphism of affine schemes determined by rational functions.

**Warning:** You should not create objects of this class directly because no type or consistency checking is performed. The preferred method to construct such dynamical systems is to use `DynamicalSystem_affine()` function.

**INPUT:**

- `morphism_or_polys` – a SchemeMorphism, a polynomial, a rational function, or a list or tuple of polynomials or rational functions
- `domain` – optional affine space or subscheme of such; the following combinations of `morphism_or_polys` and `domain` are meaningful:
  - `morphism_or_polys` is a SchemeMorphism; `domain` is ignored in this case
  - `morphism_or_polys` is a list of polynomials or rational functions that define a rational endomorphism of `domain`
  - `morphism_or_polys` is a list of polynomials or rational functions and `domain` is unspecified; `domain` is then taken to be the affine space of appropriate dimension over the common base ring, if one exists, of the elements of `morphism_or_polys`
  - `morphism_or_polys` is a single polynomial or rational function; `domain` is ignored and assumed to be the 1-dimensional affine space over the base ring of `morphism_or_polys`

**OUTPUT:** `DynamicalSystem_affine`

**EXAMPLES:**

```python
sage: A3.<x,y,z> = AffineSpace(QQ, 3)
sage: DynamicalSystem_affine([x, y, z])
Dynamical System of Affine Space of dimension 3 over Rational Field
  Defn: Defined on coordinates by sending (x, y, z) to
        (x, y, 1)

sage: R.<x,y> = QQ[

sage: DynamicalSystem_affine([x/y, y^2 + 1])
Dynamical System of Affine Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x, y) to
        (x/y, y^2 + 1)

sage: R.<t> = ZZ[

sage: DynamicalSystem_affine(t^2 - 1)
Dynamical System of Affine Space of dimension 1 over Integer Ring
  Defn: Defined on coordinates by sending (t) to
        (t^2 - 1)

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: X = A.subscheme([x-y^2])
sage: DynamicalSystem_affine([9/4*x^2, 3/2*y], domain=X)
Dynamical System of Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  -y^2 + x
  Defn: Defined on coordinates by sending (x, y) to
        (9/4*x^2, 3/2*y)
```

178 Chapter 6. Arithmetic Dynamical Systems
Notice that $\mathbb{Z}$ becomes $\mathbb{Q}$ since the function is rational:

```python
sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: DynamicalSystem_affine([3*x^2/(5*y), y^2/(2*x^2)])
```

If you pass in quotient ring elements, they are reduced:

```python
sage: A.<x,y,z> = AffineSpace(QQ, 3)
sage: X = A.subscheme([x-y])
sage: u,v,w = X.coordinate_ring().gens()
sage: DynamicalSystem_affine([u, v, u+v], domain=X)
```

```python
sage: R.<t> = PolynomialRing(QQ)
sage: A.<x,y,z> = AffineSpace(R, 3)
sage: X = A.subscheme(x^2-y^2)
sage: H = End(X)
sage: f = H([x^2/(t*y), t*y^2, x*z])
sage: DynamicalSystem_affine(f)
```

```python
sage: x = var('x')
sage: DynamicalSystem_affine(x^2+1)
```

```
Traceback (most recent call last):
  ...
TypeError: Symbolic Ring cannot be the base ring
```

**dynatomic_polynomial** *(period)*

Compute the (affine) dynatomic polynomial of a dynamical system $f : \mathbb{A}^1 \to \mathbb{A}^1$.

The dynatomic polynomial is the analog of the cyclotomic polynomial and its roots are the points of formal period $n$.

6.2. Dynamical systems on affine schemes 179
ALGORITHM:
Homogenize to a map \( f : \mathbb{P}^1 \to \mathbb{P}^1 \) and compute the dynatomic polynomial there. Then, dehomogenize.

INPUT:
- period – a positive integer or a list/tuple \([m, n]\), where \(m\) is the preperiod and \(n\) is the period

OUTPUT:
If possible, a single variable polynomial in the coordinate ring of the polynomial. Otherwise a fraction field element of the coordinate ring of the polynomial.

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: f = DynamicalSystem_affine([x^2+y^2, y^2])
sage: f.dynatomic_polynomial(2)
Traceback (most recent call last):
  ...TypeError: does not make sense in dimension >1
```

```
sage: A.<x> = AffineSpace(ZZ, 1)
sage: f = DynamicalSystem_affine([(x^2+1)/x])
sage: f.dynatomic_polynomial(4)
2*x^12 + 18*x^10 + 57*x^8 + 79*x^6 + 48*x^4 + 12*x^2 + 1
```

```
sage: A.<x> = AffineSpace(CC, 1)
sage: f = DynamicalSystem_affine([(x^2+1)/(3*x)])
sage: f.dynatomic_polynomial(3)
13.0000000000000*x^6 + 117.0000000000000*x^4 + 78.00000000000000*x^2 + 1.000000000000000
```

```
sage: A.<x> = AffineSpace(QQ, 1)
sage: f = DynamicalSystem_affine([x^2-10/9])
sage: f.dynatomic_polynomial([2, 1])
531441*x^4 - 649539*x^2 - 524880
```

```
sage: A.<x> = AffineSpace(CC, 1)
sage: f = DynamicalSystem_affine([x^2+CC.0])
sage: f.dynatomic_polynomial(2)
x^2 + x + 1.000000000000000 + 1.000000000000000*I
```

```
sage: K.<c> = FunctionField(QQ)
sage: A.<x> = AffineSpace(K, 1)
sage: f = DynamicalSystem_affine([x^2 + c])
sage: f.dynatomic_polynomial(4)
x^12 + 6*c*x^10 + x^9 + (15*c^2 + 3*c)*x^8 + 4*c*x^7 + (20*c^3 + 12*c^2 + ___1)*x^6
+ (6*c^2 + 2*c)*x^5 + (15*c^4 + 18*c^3 + 3*c^2 + 4*c)*x^4 + (4*c^3 + 4*c^2 + ___1)*x^3
+ (6*c^5 + 12*c^4 + 6*c^3 + 5*c^2 + c)*x^2 + (c^4 + 2*c^3 + c^2 + 2*c)*x
+ c^6 + 3*c^5 + 3*c^4 + 3*c^3 + 2*c^2 + 1
```

```
sage: A.<z> = AffineSpace(QQ, 1)
sage: f = DynamicalSystem_affine([z^2+3/z+1/7])
sage: f.dynatomic_polynomial(1).parent()
Multivariate Polynomial Ring in z over Rational Field
```

180 Chapter 6. Arithmetic Dynamical Systems
```
sage: R.<c> = QQ[]
sage: A.<z> = AffineSpace(R,1)
sage: f = DynamicalSystem_affine([z^2 + c])
sage: f.dynatomic_polynomial([1,1])
z^2 + z + c
```
```
sage: A.<x> = AffineSpace(CC,1)
sage: F = DynamicalSystem_affine([(1/2*x^2 + CC(sqrt(3)))])
sage: F.dynatomic_polynomial([1,1])
(0.125000000000000*x^4 + 0.366025403784439*x^2 + 1.50000000000000)/(0.500000000000000*x^2 - x + 1.73205080756888)
```

**homogenize** (*n*)

Return the homogenization of this dynamical system.

If its domain is a subscheme, the domain of the homogenized map is the projective embedding of the domain. The domain and codomain can be homogenized at different coordinates: *n*[0] for the domain and *n*[1] for the codomain.

**INPUT:**

- *n* – a tuple of nonnegative integers. If *n* is an integer, then the two values of the tuple are assumed to be the same

**OUTPUT:** *DynamicalSystem_projective*

**EXAMPLES:**

```
sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: f = DynamicalSystem_affine([(x^2-2)/x^5, y^2])
sage: f.homogenize(2)
Dynamical System of Projective Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x0 : x1 : x2) to
        (x0^2*x2^5 - 2*x2^7 : x0^5*x1^2 : x0^5*x2^2)
```

```
sage: A.<x,y> = AffineSpace(CC, 2)
sage: f = DynamicalSystem_affine([(x^2-2)/(x*y), y^2-x])
sage: f.homogenize((2, 0))
Dynamical System of Projective Space of dimension 2 over Complex Field with 53 bits of precision
  Defn: Defined on coordinates by sending (x0 : x1 : x2) to
        (x0*x1*x2^2 : x0^2*x2^2 + (-2.00000000000000000000000000)*x2^4 : x0*x1^3 - x0^2*x1*x2)
```

```
sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: X = A.subscheme([x-y^2])
sage: f = DynamicalSystem_affine([9*y^2, 3*y], domain=X)
sage: f.homogenize(2)
Dynamical System of Closed subscheme of Projective Space of dimension 2 over Integer Ring defined by:
  x1^2 - x0*x2
  Defn: Defined on coordinates by sending (x0 : x1 : x2) to
        (9*x1^2 : 3*x1*x2 : x2^2)
```

```
sage: A.<x> = AffineSpace(QQ, 1)
sage: f = DynamicalSystem_affine([x^2-1])
sage: f.homogenize((1, 0))
```

(continues on next page)
Dynamical System of Projective Space of dimension 1 over Rational Field

Defn: Defined on coordinates by sending (x0 : x1) to
(x1^2 : x0^2 - x1^2)

```python
sage: R.<a> = PolynomialRing(QQbar)
sage: A.<x,y> = AffineSpace(R, 2)
sage: f = DynamicalSystem_affine([QQbar(sqrt(2))*x*y, a*x^2])
sage: f.homogenize(2)
Dynamical System of Projective Space of dimension 2 over Univariate
Polynomial Ring in a over Algebraic Field
Defn: Defined on coordinates by sending (x0 : x1 : x2) to
(1.414213562373095?*x0*x1 : a*x0^2 : x2^2)
```

```python
sage: P.<x,y,z> = AffineSpace(QQ, 3)
sage: f = DynamicalSystem_affine([x^2 - 2*x*y + z*x, z^2 - y^2 , 5*z*y])
sage: f.homogenize(2).dehomogenize(2) == f
True
```

**multiplier** \((P, n, check=True)\)

Return the multiplier of the point \(P\) of period \(n\) by this dynamical system.

**INPUT:**

- \(P\) – a point on domain of the map
- \(n\) – a positive integer, the period of \(P\)
- \(check\) – (default: True) boolean, verify that \(P\) has period \(n\)

**OUTPUT:**

A square matrix of size \(\text{self.codomain().dimension_relative()}\) in the \(\text{base_ring}\) of the map.

**EXAMPLES:**

```python
sage: P.<x,y> = AffineSpace(QQ, 2)
sage: f = DynamicalSystem_affine([x^2, y^2])
sage: f.multiplier(P([1, 1]), 1)
[2 0]
[0 2]
```

```python
sage: P.<x,y,z> = AffineSpace(QQ, 3)
sage: f = DynamicalSystem_affine([x^2 - 2*x*y + z*x, z^2 - y^2 , 5*z*y])
sage: f.multiplier(P([1/2, 1, 0]), 2)
[1 0 0]
[0 4 0]
[0 0 0]
```

```python
sage: P.<x> = AffineSpace(CC, 1)
sage: f = DynamicalSystem_affine([x^2 + 1/2])
sage: f.multiplier(P([0.5 + 0.5*I]), 1)
[1.00000000000000 + 1.00000000000000*I]
```

```python
sage: R.<t> = PolynomialRing(CC, 1)
sage: P.<x> = AffineSpace(R, 1)
sage: f = DynamicalSystem_affine([x^2 - t^2 + t])
```

(continues on next page)
sage: f.multiplier(P([-t + 1]), 1)
[(-2.00000000000000)*t + 2.00000000000000]
sage: P.<x,y> = AffineSpace(QQ, 2)
sage: X = P.subscheme([x^2-y^2])
sage: f = DynamicalSystem_affine([x^2, y^2], domain=X)
sage: f.multiplier(X([1, 1]), 1)
[2 0]
[0 2]

nth_iterate \((P, n)\)
Return the \(n\)-th iterate of the point \(P\) by this dynamical system.

INPUT:

- \(P\) – a point in the map’s domain
- \(n\) – a positive integer

OUTPUT: a point in the map’s codomain

EXAMPLES:

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: f = DynamicalSystem_affine([(x-2*y^2)/x, 3*x*y])
sage: f.nth_iterate(A(9, 3), 3)
(-104975/13123, -9566667)
sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: X = A.subscheme([x-y^2])
sage: f = DynamicalSystem_affine([9*y^2, 3*y], domain=X)
sage: f.nth_iterate(X(9, 3), 4)
(59049, 243)
sage: R.<t> = PolynomialRing(QQ)
sage: A.<x,y> = AffineSpace(FractionField(R), 2)
sage: f = DynamicalSystem_affine([(x-t*y^2)/x, t*x*y])
sage: f.nth_iterate(A(1, t), 3)
((-t^16 + 3*t^13 - 3*t^10 + t^7 + t^5 + t^3 - 1)/(t^5 + t^3 - 1), -t^9 - t^7 + t^4)
sage: A.<x,y,z> = AffineSpace(QQ, 3)
sage: X = A.subscheme([x^2-y^2])
sage: f = DynamicalSystem_affine([x^2, y^2, x+y], domain=X)
sage: f.nth_iterate_map(2)
Dynamical System of Closed subscheme of Affine Space of dimension 3 over Rational Field defined by:
\(x^2 - y^2\)
Defn: Defined on coordinates by sending \((x, y, z)\) to
\((x^4, y^4, x^2 + y^2)\)

nth_iterate_map \((n)\)
Return the \(n\)-th iterate of self.

ALGORITHM:

Uses a form of successive squaring to reducing computations.
Todo: This could be improved.

INPUT:

- n – a positive integer

OUTPUT: a dynamical system of affine space

EXAMPLES:

```python
sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: f = DynamicalSystem_affine([(x^2-2)/(2*y), y^2-3*x])
sage: f.nth_iterate_map(2)
Dynamical System of Affine Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x, y) to
        ((x^4 - 4*x^2 - 8*y^2 + 4)/(8*y^4 - 24*x*y^2), (2*y^5 - 12*x*y^3
          + 18*x^2*y - 3*x^2 + 6)/(2*y))
```

```python
sage: A.<x> = AffineSpace(QQ, 1)
sage: f = DynamicalSystem_affine([(3*x^2-2)/(x)])
sage: f.nth_iterate_map(3)
Dynamical System of Affine Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x) to
        ((2187*x^8 - 6174*x^6 + 6300*x^4 - 2744*x^2 + 432)/(81*x^7 -
          168*x^5 + 112*x^3 - 24*x))
```

```python
sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: X = A.subscheme([x-y^2])
sage: f = DynamicalSystem_affine([9*x^2, 3*y], domain=X)
sage: f.nth_iterate_map(2)
Dynamical System of Closed subscheme of Affine Space of dimension 2
  over Integer Ring defined by:
    -y^2 + x
  Defn: Defined on coordinates by sending (x, y) to
        (729*x^4, 9*y)
```

```python
sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: f = DynamicalSystem_affine([(3/5*x^2, y^2/(2*x^2))])
sage: f.nth_iterate_map(2)
Dynamical System of Affine Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x, y) to
        (27/125*x^4, y^4/(72/25*x^8))
```

orbit \((P, n)\)

Return the orbit of \(P\) by the dynamical system.

Let \(F\) be this dynamical system. If \(n\) is an integer return \([P, F(P), \ldots, F^n(P)]\). If \(n\) is a list or tuple \(n = [m, k]\) return \([F^m(P), \ldots, F^k(P)]\).

INPUT:

- \(P\) – a point in the map’s domain
- \(n\) – a non-negative integer or list or tuple of two non-negative integers

OUTPUT: a list of points in the map’s codomain

EXAMPLES:
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: f = DynamicalSystem_affine([(x-2*y^2)/x, 3*x+y])
sage: f.orbit(A(9, 3), 3)
[(9, 3), (-1, 81), (13123, -243), (-104975/13123, -9566667)]

sage: A.<x> = AffineSpace(QQ, 1)
sage: f = DynamicalSystem_affine([(x-2)/x])
sage: f.orbit(A(1/2), [1, 3])
[(-3), (5/3), (-1/5)]

sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: X = A.subscheme([x-y^2])
sage: f = DynamicalSystem_affine([9*y^2, 3*y], domain=X)
sage: f.orbit(X(9, 3), (0, 4))
[(9, 3), (81, 9), (729, 27), (6561, 81), (59049, 243)]

sage: R.<t> = PolynomialRing(QQ)
sage: A.<x,y> = AffineSpace(FractionField(R), 2)
sage: f = DynamicalSystem_affine([(x-t*y^2)/x, t*x*y])
sage: f.orbit(A(1, t), 3)
[(1, t),
 (-t^3 + 1, t^2),
 (t^5 + t^3 - 1)/(t^3 - 1), -t^6 + t^3),
 ((-t^16 + 3*t^13 - 3*t^10 + t^7 + t^5 + t^3 - 1)/(t^5 + t^3 - 1), -t^9 - t^7 + t^4)]

class sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine_field(polys_or_rat_fncts, domain)

Bases: sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine, sage.schemes.affine.affine_morphism.SchemeMorphism_polynomial_affine_space_field

weil_restriction()

Compute the Weil restriction of this morphism over some extension field.

If the field is a finite field, then this computes the Weil restriction to the prime subfield.

A Weil restriction of scalars - denoted \( R_{L/k} \) - is a functor which, for any finite extension of fields \( L/k \) and any algebraic variety \( X \) over \( L \), produces another corresponding variety \( R_{L/k}(X) \), defined over \( k \).

It is useful for reducing questions about varieties over large fields to questions about more complicated varieties over smaller fields. Since it is a functor it also applied to morphisms. In particular, the functor applied to a morphism gives the equivalent morphism from the Weil restriction of the domain to the Weil restriction of the codomain.

OUTPUT:

Scheme morphism on the Weil restrictions of the domain and codomain of the map.

EXAMPLES:

sage: K.<v> = QuadraticField(5)
sage: A.<x,y> = AffineSpace(K, 2)
sage: f = DynamicalSystem_affine([x^2-y^2, y^2])
sage: f.weil_restriction()
Dynamical System of Affine Space of dimension 4 over Rational Field
Defn: Defined on coordinates by sending (z0, z1, z2, z3) to
\( (z0^2 + 5*z1^2 - z2^2 - 5*z3^2, 2*z0*z1 - 2*z2*z3, z2^2 + 5*z3^2, \ldots) \)
sage: K.<v> = QuadraticField(5)
sage: PS.<x,y> = AffineSpace(K, 2)
sage: f = DynamicalSystem_affine([x, y])
sage: F = f.weil_restriction()
sage: P = PS(2, 1)
sage: Q = P.weil_restriction()
sage: f(P).weil_restriction() == F(Q)
True

class sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine_finite_field

Bases: sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine_field,
sage.schemes.affine.affine_morphism.SchemeMorphism_polynomial_affine_space_finite_field

cyclegraph()

Return the digraph of all orbits of this morphism mod \( p \).

For subschemes, only points on the subscheme whose image are also on the subscheme are in the digraph.

OUTPUT: a digraph

EXAMPLES:

```python
sage: P.<x,y> = AffineSpace(GF(5), 2)
sage: f = DynamicalSystem_affine([x^2-y, x*y+1])
sage: f.cyclegraph()
Looped digraph on 25 vertices
sage: P.<x> = AffineSpace(GF(3^3, 't'), 1)
sage: f = DynamicalSystem_affine([x^2-1])
sage: f.cyclegraph()
Looped digraph on 27 vertices
sage: P.<x,y> = AffineSpace(GF(7), 2)
sage: X = P.subscheme(x-y)
sage: f = DynamicalSystem_affine([x^2, y^2], domain=X)
sage: f.cyclegraph()
Looped digraph on 7 vertices
```

orbit_structure(\( P \))

Every point is preperiodic over a finite field.

This function returns the pair \([m, n]\) where \( m \) is the preperiod and \( n \) is the period of the point \( P \) by this map.

INPUT:

- \( P \) – a point in the map’s domain

OUTPUT: a list \([m, n]\) of integers

EXAMPLES:
6.3 Dynamical systems on projective schemes

A dynamical system of projective schemes determined by homogeneous polynomials functions that define what the morphism does on points in the ambient projective space.

The main constructor functions are given by DynamicalSystem and DynamicalSystem_projective. The constructors function can take either polynomials or a morphism from which to construct a dynamical system. If the domain is not specified, it is constructed. However, if you plan on working with points or subvarieties in the domain, it recommended to specify the domain.

The initialization checks are always performed by the constructor functions. It is possible, but not recommended, to skip these checks by calling the class initialization directly.

AUTHORS:

• David Kohel, William Stein
• William Stein (2006-02-11): fixed bug where P(0,0,0) was allowed as a projective point.
• Volker Braun (2011-08-08): Renamed classes, more documentation, misc cleanups.
• Ben Hutz (2013-03) iteration functionality and new directory structure for affine/projective, height functionality
• Brian Stout, Ben Hutz (Nov 2013) - added minimal model functionality
• Dillon Rose (2014-01): Speed enhancements
• Ben Hutz (2015-11): iteration of subschemes
• Ben Hutz (2017-7): relocate code and create class

class sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective(polys, domain)

Bases: sage.schemes.projective.projective_morphism.SchemeMorphism_polynomial_projective_space,
sage.dynamics.arithmetic_dynamics.generic_ds.DynamicalSystem

A dynamical system of projective schemes determined by homogeneous polynomials that define what the morphism does on points in the ambient projective space.

Warning: You should not create objects of this class directly because no type or consistency checking is performed. The preferred method to construct such dynamical systems is to use DynamicalSystem_projective() function

INPUT:

• morphism_or_polys – a SchemeMorphism, a polynomial, a rational function, or a list or tuple of homogeneous polynomials.
• domain – optional projective space or projective subscheme.

• names – optional tuple of strings to be used as coordinate names for a projective space that is constructed; defaults to 'X', 'Y'.

The following combinations of morphism_or_polys and domain are meaningful:

– morphism_or_polys is a SchemeMorphism; domain is ignored in this case.

– morphism_or_polys is a list of homogeneous polynomials that define a rational endomorphism of domain.

– morphism_or_polys is a list of homogeneous polynomials and domain is unspecified; domain is then taken to be the projective space of appropriate dimension over the common base ring, if one exists, of the elements of morphism_or_polys.

– morphism_or_polys is a single polynomial or rational function; domain is ignored and taken to be a 1-dimensional projective space over the base ring of morphism_or_polys with coordinate names given by names.

OUTPUT: DynamicalSystem_projective.

EXAMPLES:

```
sage: P1.<x,y> = ProjectiveSpace(QQ,1)
sage: DynamicalSystem_projective([y, 2*x])
Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to
       (y : 2*x)
```

We can define dynamical systems on \( P^1 \) by giving a polynomial or rational function:

```
sage: R.<t> = QQ[]
sage: DynamicalSystem_projective(t^2 - 3)
Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (X : Y) to
       (X^2 - 3*Y^2 : Y^2)
sage: DynamicalSystem_projective(1/t^2)
Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (X : Y) to
       (Y^2 : X^2)
sage: R.<x> = PolynomialRing(QQ,1)
sage: DynamicalSystem_projective(x^2, names=['a','b'])
Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (a : b) to
       (a^2 : b^2)
```

Symbolic Ring elements are not allowed:

```
sage: x,y = var('x,y')
sage: DynamicalSystem_projective([x^2,y^2])
Traceback (most recent call last):
...    ValueError: [x^2, y^2] must be elements of a polynomial ring
```

```
sage: R.<x> = PolynomialRing(QQ,1)
sage: DynamicalSystem_projective(x^2)
Dynamical System of Projective Space of dimension 1 over Rational Field
```

(continues on next page)
Defn: Defined on coordinates by sending $(X : Y)$ to $(X^2 : Y^2)$

```
sage: R.<t> = PolynomialRing(QQ)
sage: P.<x,y,z> = ProjectiveSpace(R, 2)
sage: X = P.subscheme([x])
sage: DynamicalSystem_projective([x^2, t*y^2, x*z], domain=X)
```

Dynamical System of Closed subscheme of Projective Space of dimension 2 over Univariate Polynomial Ring in t over Rational Field defined by:

$x$

Defn: Defined on coordinates by sending $(x : y : z)$ to $(x^2 : t*y^2 : x*z)$

When elements of the quotient ring are used, they are reduced:

```
sage: P.<x,y,z> = ProjectiveSpace(CC, 2)
sage: X = P.subscheme([x-y])
sage: u,v,w = X.coordinate_ring().gens()
sage: DynamicalSystem_projective([u^2, v^2, w*u], domain=X)
```

Dynamical System of Closed subscheme of Projective Space of dimension 2 over Complex Field with 53 bits of precision defined by:

$x - y$

Defn: Defined on coordinates by sending $(x : y : z)$ to $(y^2 : y^2 : y*z)$

We can also compute the forward image of subschemes through elimination. In particular, let $X = V(h_1, ..., h_t)$ and define the ideal $I = (h_1, ..., h_t, y_0 - f_0(\bar{x}), ..., y_n - f_n(\bar{x}))$. Then the elimination ideal $I_{n+1} = I \cap K[y_0, ..., y_n]$ is a homogeneous ideal and $f(X) = V(I_{n+1})$:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([(x-2*y)^2, (x-2*z)^2, x^2])
sage: X = P.subscheme(y-z)
sage: f(f(f(X)))
```

Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:

$y - z$

```
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: f = DynamicalSystem_projective([(x-2*y)^2, (x-2*z)^2, (x-2*w)^2, x^2])
sage: f(P.subscheme([x, y, z]))
```

Closed subscheme of Projective Space of dimension 3 over Rational Field defined by:

$w, y, x$

```
sage: T.<x,y,z,w,u> = ProductProjectiveSpaces([2, 1], QQ)
sage: DynamicalSystem_projective([x^2*u, y^2*w, z^2*u, w^2, u^2], domain=T)
```

Dynamical System of Product of projective spaces $P^2 \times P^1$ over Rational Field

Defn: Defined by sending $(x : y : z : w : u)$ to $(x^2*u : y^2*w : z^2*u, w^2 : u^2)$.

```
all_minimal_models(return_transformation=False, prime_list=None, algorithm=None, check_minimal=True)
```

Determine a representative in each $SL(2,Z)$-orbit of this map.

6.3. Dynamical systems on projective schemes 189
This can be done either with the Bruin-Molnar algorithm or the Hutz-Stoll algorithm. The Hutz-Stoll algorithm requires the map to have minimal resultant and then finds representatives in orbits with minimal resultant. The Bruin-Molnar algorithm finds representatives with the same resultant (up to sign) of the given map.

Bruin-Molnar does not work for polynomials and is more efficient for large primes.

**INPUT:**
- `return_transformation` – (default: `False`) boolean; this signals a return of the $PGL_2$ transformation to conjugate this map to the calculated models
- `prime_list` – (optional) a list of primes, in case one only wants to determine minimality at those specific primes
- `algorithm` – (optional) string; can be one of the following:
  - `BM` - the Bruin-Molnar algorithm [BM2012]
  - `HS` - for the Hutz-Stoll algorithm [HS2018]
  if not specified, properties of the map are utilized to choose
- `check_minimal` – (optional) boolean; to first check if the map is minimal and if not, compute a minimal model before computing for orbit representatives

**OUTPUT:**
A list of pairs $(F, m)$, where $F$ is dynamical system on the projective line and $m$ is the associated $PGL(2, \mathbb{Q})$ element. Or just a list of dynamical systems if not returning the conjugation.

**EXAMPLES:**

```python
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem([2*x^2, 3*y^2])
sage: f.all_minimal_models()
[Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to
  (x^2 : y^2)]
```

```python
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: c = 2*3^6
sage: f = DynamicalSystem([x^3 - c^2*y^3, x*y^2])
sage: len(f.all_minimal_models(algorithm='HS'))
14
sage: len(f.all_minimal_models(prime_list=[2], algorithm='HS'))
2
```

**REFERENCES:**
- [BM2012]
- [HS2018]
The automorphism group is the set of $PGL(2)$ elements that fixes this map under conjugation.

INPUT:

keywords:
• starting_prime – (default: 5) the first prime to use for CRT
• algorithm – (optional) can be one of the following:
  – 'CRT' - Chinese Remainder Theorem
  – 'fixed_points' - fixed points algorithm
• return_functions – (default: False) boolean; True returns elements as linear fractional transformations and False returns elements as $PGL2$ matrices
• iso_type – (default: False) boolean; True returns the isomorphism type of the automorphism group

OUTPUT: a list of elements in the automorphism group

AUTHORS:
• Original algorithm written by Xander Faber, Michelle Manes, Bianca Viray
• Modified by Joao Alberto de Faria, Ben Hutz, Bianca Thompson

REFERENCES:
• [FMV2014]

EXAMPLES:

```
sage: R.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2-y^2, x*y])
sage: f.automorphism_group(return_functions=True)
[x, -x]
```

```
sage: R.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 + 5*x*y + 5*y^2, 5*x^2 + 5*x*y + y^2])
sage: f.automorphism_group()
[ [1 0] [0 2]
  [0 1], [2 0] ]
```

```
sage: R.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2-2*x*y-2*y^2, -2*x^2-2*x*y+y^2])
sage: f.automorphism_group(return_functions=True)
[x, 1/x, -x - 1, -x/(x + 1), (-x - 1)/x, -1/(x + 1)]
```

```
sage: R.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([3*x^2*y - y^3, x^3 - 3*x*y^2])
sage: f.automorphism_group(algorithm='CRT', return_functions=True, iso_type=True)
([(x, (x + 1)/(x - 1)), (-x + 1)/(x + 1), -x, 1/x, -1/x, (x - 1)/(x + 1), (-x - 1)/(x - 1)], 'Dihedral of order 8')
```

6.3. Dynamical systems on projective schemes 191
sage: A.<z> = AffineSpace(QQ,1)
sage: f = DynamicalSystem_affine([1/z^3])
sage: F = f.homogenize(1)
sage: F.automorphism_group()
[ [1 0] [0 2] [-1 0] [ 0 -2]
[0 1], [2 0], [ 0 1], [ 2 0]]

sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: f = DynamicalSystem_projective([x^2 + x*z, y^2, z^2])
sage: f.automorphism_group() # long time
[ [1 0 0]
[0 1 0]
[0 0 1]]

sage: K.<w> = CyclotomicField(3)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: D6 = DynamicalSystem_projective([y^2,x^2])
sage: D6.automorphism_group()
[ [1 0] [0 w] [0 1] [-w - 1 0] [ 0 -w - 1]
[0 1], [1 0], [1 0], [0 1], [ 0 1], [ 1 0]]

canonical_height $(P, **kwds)$
Evaluate the (absolute) canonical height of $P$ with respect to this dynamical system.

Must be over number field or order of a number field. Specify either the number of terms of the series to evaluate or the error bound required.

ALGORITHM:
The sum of the Green’s function at the archimedean places and the places of bad reduction.
If function is defined over $\mathbb{Q}$ uses Wells’ Algorithm, which allows us not to have to factor the resultant.

INPUT:

• $P$ – a projective point

kwds:

• `badprimes` – (optional) a list of primes of bad reduction
• $N$ – (default: 10) positive integer. number of terms of the series to use in the local green functions
• `prec` – (default: 100) positive integer, float point or $p$-adic precision
• `error_bound` – (optional) a positive real number

OUTPUT: a real number

EXAMPLES:

sage: P.<x,y> = ProjectiveSpace(ZZ,1)
sage: f = DynamicalSystem_projective([x^2+y^2, 2*x*y]);
sage: f.canonical_height(P.point([5,4]), error_bound=0.001)
2.1970553519503404898926835324
Notice that preperiodic points may not return exactly 0:

```python
sage: R.<X> = PolynomialRing(QQ)
sage: K.<a> = NumberField(X^2 + X - 1)
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x^2-2*y^2, y^2])
sage: Q = P.point([a,1])
sage: f.canonical_height(Q, error_bound=0.000001)  # Answer only within error_bound of 0
2.736491978870160119266380480e-8
sage: f.nth_iterate(Q,2) == Q  # but it is indeed preperiodic
True
```

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: X = P.subscheme(x^2-y^2);
sage: f = DynamicalSystem_projective([x^2, y^2, 4*z^2], domain=X);
sage: Q = X([4,4,1])
sage: f.canonical_height(Q, badprimes=[2])
0.0013538030870311431824555314882
```

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: X = P.subscheme(x^2-y^2);
sage: f = DynamicalSystem_projective([x^2, y^2, 30*z^2], domain=X)
sage: Q = X([4, 4, 1])
sage: f.canonical_height(Q, badprimes=[2,3,5], prec=200)
2.7054056208276961889745202199786469356774912979228770208655455481
```

```python
sage: RSA768
1230186684530117755130494958384962707285356959533479219732245215
172640050726365751874520219978646938995647942774063845925192557326303453731548
268507917026122142913461670429214311602221240479274737794080665351419574598569
02143413
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([1000*x^2-29*y^2, 1000*y^2])
sage: Q = P(-1/4, 1)
sage: f.canonical_height(Q, error_bound=0.01)
3.79607979254623065837411853
```

```python
sage: RSA768
0x<1230186684530117755130494958384962707285356959533479219732245215>
0x<172640050726365751874520219978646938995647942774063845925192557326303453731548>
0x<268507917026122142913461670429214311602221240479274737794080665351419574598569>
0x<02143413>
```

```python
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([RSA768*x^2 + y^2, x*y])
sage: Q = P(RSA768,1)
sage: f.canonical_height(Q, error_bound=0.00000000000000001)
931.1825642271842178672729195
```

```python
sage: P.<x,y>=ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem([2*x*(-2*x^3 + 3*(x^2*y)) + 3*y^3],domain=QQ)
sage: f.canonical_height(P(1,0))
0.000000000000000000000000000000000000000
```

### 6.3. Dynamical systems on projective schemes
Conjugate this dynamical system by $M$, i.e. $M^{-1} \circ f \circ M$.

If possible the new map will be defined over the same space. Otherwise, will try to coerce to the base ring of $M$.

**INPUT:**

- $M$ – a square invertible matrix

**OUTPUT:** a dynamical system

**EXAMPLES:**

```python
sage: P.<x,y> = ProjectiveSpace(ZZ,1)
sage: f = DynamicalSystem_projective([x^2+y^2, y^2])
sage: f.conjugate(matrix([[1,2], [0,1]]))
Dynamical System of Projective Space of dimension 1 over Integer Ring
  Defn: Defined on coordinates by sending (x : y) to
  (x^2 + 4*x*y + 3*y^2 : y^2)
```

```python
sage: R.<x> = PolynomialRing(QQ)
sage: K.<i> = NumberField(x^2+1)
sage: P.<x,y> = ProjectiveSpace(ZZ,1)
sage: f = DynamicalSystem_projective([x^3+y^3, y^3])
sage: f.conjugate(matrix([[i,0], [0,-i]]))
Dynamical System of Projective Space of dimension 1 over Integer Ring
  Defn: Defined on coordinates by sending (x : y) to
  (-x^3 + y^3 : -y^3)
```

```python
sage: P.<x,y,z> = ProjectiveSpace(ZZ,2)
sage: f = DynamicalSystem_projective([x^2+y^2 ,y^2, y*z])
sage: f.conjugate(matrix([[1,2,3], [0,1,2], [0,0,1]]))
Dynamical System of Projective Space of dimension 2 over Integer Ring
  Defn: Defined on coordinates by sending (x : y : z) to
  (x^2 + 4*x*y + 3*y^2 + 6*x*z + 9*y*z + 7*z^2 : y^2 + 2*y*z : y*z +
  2*z^2)
```

```python
sage: R.<x> = PolynomialRing(QQ)
sage: K.<i> = NumberField(x^2+1)
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([1/3*x^2+1/2*y^2, y^2])
sage: f.conjugate(matrix([[i,0], [0,-i]]))
Dynamical System of Projective Space of dimension 1 over Number Field in i
  with defining polynomial x^2 + 1
  Defn: Defined on coordinates by sending (x : y) to
  ((1/3*i)*x^2 + (1/2*i)*y^2 : (-i)*y^2)
```

**critical_height (**)kwd**

Compute the critical height of this dynamical system.

The critical height is defined by J. Silverman as the sum of the canonical heights of the critical points. This must be dimension 1 and defined over a number field or number field order.
INPUT:

kwds:
  • badprimes – (optional) a list of primes of bad reduction
  • N – (default: 10) positive integer; number of terms of the series to use in the local green functions
  • prec – (default: 100) positive integer, float point or \( p \)-adic precision
  • error_bound – (optional) a positive real number
  • embedding – (optional) the embedding of the base field to \( \mathbb{Q} \)

OUTPUT: real number

EXAMPLES:

```sage
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^3+7*y^3, 11*y^3])
sage: f.critical_height()
1.1989273321156851418802151128
```

```sage
sage: K.<w> = QuadraticField(2)
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x^2+w*y^2, y^2])
sage: f.critical_height()
0.16090842452312941163719755472
```

Postcritically finite maps have critical height 0:

```sage
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^3-3/4*x*y^2 + 3/4*y^3, y^3])
sage: f.critical_height(error_bound=0.0001)
0.00000000000000000000000000000
```

```sage
critical_point_portrait(check=True, embedding=None)
```

This is the directed graph of iterates starting with the critical points. Must be dimension 1. If \texttt{check} is True, then the map is first checked to see if it is postcritically finite.

INPUT:

• check – boolean

• embedding – embedding of base ring into \( \mathbb{Q} \)

OUTPUT: a digraph

EXAMPLES:

```sage
sage: R.<z> = QQ[]
sage: K.<v> = NumberField(z^6 + 2*z^5 + 2*z^4 + 2*z^3 + z^2 + 1)
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x^2+v*y^2, y^2])
sage: f.critical_point_portrait(check=False, embedding=K.˓
˓embeddings(QQbar)[0]) # long time
Looped digraph on 6 vertices
```
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^5 + 5/4*x*y^4, y^5])
sage: f.critical_point_portrait(check=False)
Looped digraph on 5 vertices

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 + 2*y^2, y^2])
sage: f.critical_point_portrait()
Traceback (most recent call last):
...
TypeError: map must be post-critically finite

**critical_points** \((R=None)\)

Return the critical points of this dynamical system defined over the ring \(R\) or the base ring of this map.

Must be dimension 1.

**INPUT:**

- \(R\) – (optional) a ring

**OUTPUT:** a list of projective space points defined over \(R\)

**EXAMPLES:**

```python
sage: set_verbose(None)
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^3-2*x*y^2 + 2*y^3, y^3])
sage: f.critical_points()
[(1 : 0)]
sage: K.<w> = QuadraticField(6)
sage: f.critical_points(K)
[(-1/3*w : 1), (1/3*w : 1), (1 : 0)]
sage: set_verbose(None)
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([2*x^2-y^2, x*y])
sage: f.critical_points(QQbar)
[(-0.7071067811865475?*I : 1), (0.7071067811865475?*I : 1)]
```

**critical_subscheme** ()

Return the critical subscheme of this dynamical system.

**OUTPUT:** projective subscheme

**EXAMPLES:**

```python
sage: set_verbose(None)
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^3-2*x*y^2 + 2*y^3, y^3])
sage: f.critical_subscheme()
Closed subscheme of Projective Space of dimension 1 over Rational Field defined by:
9*x^2*y^2 - 6*y^4
```

(continues on next page)
(continued from previous page)

```
sage: f.critical_subscheme()
Closed subscheme of Projective Space of dimension 1 over Rational Field
defined by:
4*x^2 + 2*y^2

sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: f = DynamicalSystem_projective([2*x^2-y^2, x+y, z^2])
sage: f.critical_subscheme()
Closed subscheme of Projective Space of dimension 2 over Rational Field
defined by:
8*x^2*z + 4*y^2*z

sage: P.<x,y,z,w> = ProjectiveSpace(GF(81),3)
sage: g = DynamicalSystem_projective([x^3+y^3, y^3+z^3, z^3+x^3, w^3])
sage: g.critical_subscheme()
Closed subscheme of Projective Space of dimension 3 over Finite Field in
z4 of size 3^4 defined by:
0

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2,x*y])
sage: f.critical_subscheme()
Traceback (most recent call last):
  ...
TypeError: the function is not a morphism
```

degree_sequence (iterates=2)
Return sequence of degrees of normalized iterates starting with the
degree of this dynamical system.

INPUT: iterates – (default: 2) positive integer

OUTPUT: list of integers

EXAMPLES:

```
sage: P2.<X,Y,Z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([Z^2, X*Y, Y^2])
sage: f.degree_sequence(15)
[2, 3, 5, 8, 11, 17, 24, 31, 45, 56, 68, 91, 93, 184, 275]

sage: F.<t> = PolynomialRing(QQ)
sage: P2.<X,Y,Z> = ProjectiveSpace(F, 2)
sage: f = DynamicalSystem_projective([Y*Z, X*Y, Y^2 + t*X*Z])
sage: f.degree_sequence(5)
[2, 3, 5, 8, 13]

sage: P2.<X,Y,Z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([X^2, Y^2, Z^2])
sage: f.degree_sequence(10)
[2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]

sage: P2.<X,Y,Z> = ProjectiveSpace(ZZ, 2)
sage: f = DynamicalSystem_projective([X*Y, Y*Z+Z^2, Z^2])
sage: f.degree_sequence(10)
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

6.3. Dynamical systems on projective schemes
**dehomogenize** \((n)\)

Return the standard dehomogenization at the \(n[0]\) coordinate for the domain and the \(n[1]\) coordinate for the codomain.

Note that the new function is defined over the fraction field of the base ring of this map.

**INPUT:**

- \(n\) – a tuple of nonnegative integers; if \(n\) is an integer, then the two values of the tuple are assumed to be the same

**OUTPUT:**

If the dehomogenizing indices are the same for the domain and codomain, then a *DynamicalSystem_affine* given by dehomogenizing the source and target of \(self\) with respect to the given indices, is returned. If the dehomogenizing indices for the domain and codomain are different then the resulting affine patches are different and a scheme morphism is returned.

**EXAMPLES:**

```python
sage: P.<x,y> = ProjectiveSpace(ZZ,1)
sage: f = DynamicalSystem_projective([x^2+y^2, y^2])
sage: f.dehomogenize(0)
Dynamical System of Affine Space of dimension 1 over Integer Ring
Defn: Defined on coordinates by sending (y) to
\(y^2/(y^2 + 1)\)
sage: f.dehomogenize((0, 1))
Scheme morphism:
From: Affine Space of dimension 1 over Integer Ring
To: Affine Space of dimension 1 over Integer Ring
Defn: Defined on coordinates by sending (y) to
\((y^2 + 1)/y^2)\)
```

**dynamical_degree** \((N=3, \text{prec}=53)\)

Return an approximation to the dynamical degree of this dynamical system. The dynamical degree is defined as \(\lim_{n \to \infty} \sqrt[n]{\deg(f^n)}\).

**INPUT:**

- \(N\) – (default: 3) positive integer, iterate to use for approximation
- \(\text{prec}\) – (default: 53) positive integer, real precision to use when computing root

**OUTPUT:** real number

**EXAMPLES:**

```python
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2 + (x*y), y^2])
sage: f.dynamical_degree()
2.00000000000000
sage: P2.<X,Y,Z> = ProjectiveSpace(ZZ, 2)
sage: f = DynamicalSystem_projective([X*Y, Y*Z+Z^2, Z^2])
sage: f.dynamical_degree(N=5, prec=100)
1.43096908110525
```

**dynatomic Polynomial** \((\text{period})\)

For a dynamical system of \(\mathbb{P}^1\) compute the dynatomic polynomial.

The dynatomic polynomial is the analog of the cyclotomic polynomial and its roots are the points of formal period \(\text{period}\). If possible the division is done in the coordinate ring of this map and a polynomial
is returned. In rings where that is not possible, a FractionField element will be returned. In certain cases, when the conversion back to a polynomial fails, a SymbolicRing element will be returned.

ALGORITHM:
For a positive integer \( n \), let \([F_n, G_n]\) be the coordinates of the \( n\)th iterate of \( f \). Then construct

\[
\Phi^*_n(f)(x, y) = \sum_{d|n} (yF_d(x, y) - xG_d(x, y)) \mu(n/d),
\]

where \( \mu \) is the Möbius function.

For a pair \([m, n]\), let \( f^m = [F_m, G_m] \). Compute

\[
\Phi^*_{m,n}(f)(x, y) = \Phi^*_n(f)(F_m, G_m)/\Phi^*_n(f)(F_{m-1}, G_{m-1})
\]

REFERENCES:
- [Hutz2015]
- [MoPa1994]

INPUT:
- period – a positive integer or a list/tuple \([m, n]\) where \( m \) is the preperiod and \( n \) is the period

OUTPUT:
If possible, a two variable polynomial in the coordinate ring of this map. Otherwise a fraction field element of the coordinate ring of this map. Or, a SymbolicRing element.

Todo:
- Do the division when the base ring is \( p \)-adic so that the output is a polynomial.
- Convert back to a polynomial when the base ring is a function field (not over \( \mathbb{Q} \) or \( \mathbb{F}_p \)).

EXAMPLES:

```python
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 + y^2, y^2])
sage: f.dynatomic_polynomial(2)
x^2 + x*y + 2*y^2
```

```python
sage: P.<x,y> = ProjectiveSpace(ZZ,1)
sage: f = DynamicalSystem_projective([x^2 + y^2, x*y])
sage: f.dynatomic_polynomial(4)
2*x^12 + 18*x^10*y^2 + 57*x^8*y^4 + 79*x^6*y^6 + 48*x^4*y^8 + 12*x^2*y^10 + y^12
```

```python
sage: P.<x,y> = ProjectiveSpace(CC,1)
sage: f = DynamicalSystem_projective([x^2 + y^2, 3*x*y])
sage: f.dynatomic_polynomial(3)
13.0000000000000*x^6 + 117.0000000000000*x^4*y^2 + 78.0000000000000*x^2*y^4 + y^6
```

```python
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 - 10/9*y^2, y^2])
sage: f.dynatomic_polynomial([2,1])
x^4*y^2 - 11/9*x^2*y^4 - 80/81*y^6
```

6.3. Dynamical systems on projective schemes 199
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 - 29/16*y^2, y^2])
sage: f.dynatomic_polynomial([2,3])
x^12 - 95/8*x^10*y^2 + 13799/256*x^8*y^4 - 119953/1024*x^6*y^6 +
8198847/65536*x^4*y^8 - 31492431/524288*x^2*y^10 +
172692729/16777216*y^12

sage: P.<x,y> = ProjectiveSpace(ZZ,1)
sage: f = DynamicalSystem_projective([x^2 - y^2, y^2])
sage: f.dynatomic_polynomial([1,2])
x^2 - x*y

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^3 - y^3, 3*x*y^2])
sage: f.dynatomic_polynomial([0,4])==f.dynatomic_polynomial(4)
True

sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: f = DynamicalSystem_projective([x^2 + y^2, x*y, z^2])
sage: f.dynatomic_polynomial(2)
Traceback (most recent call last):
...
TypeError: does not make sense in dimension >1

sage: P.<x,y> = ProjectiveSpace(Qp(5),1)
sage: f = DynamicalSystem_projective([x^2 + y^2, y^2])
sage: f.dynatomic_polynomial(2)
(x^4*y + (2 + O(5^20))*x^2*y^3 - x*y^4 + (2 + O(5^20))*y^5)/(x^2*y -
x*y^2 + y^3)

sage: L.<t> = PolynomialRing(QQ)
sage: P.<x,y> = ProjectiveSpace(L,1)
sage: f = DynamicalSystem_projective([x^2 + t*y^2, y^2])
sage: f.dynatomic_polynomial(2)
x^2 + x*y + (t + 1)*y^2

sage: K.<c> = PolynomialRing(ZZ)
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x^2 + c*y^2, y^2])
sage: f.dynatomic_polynomial([1, 2])
x^2 - x*y + (c + 1)*y^2

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 + y^2, y^2])
sage: f.dynatomic_polynomial(2)
x^2 + x*y + 2*y^2

sage: R.<X> = PolynomialRing(QQ)
sage: K.<c> = NumberField(X^2 + X + 2)
sage: PP = P.change_ring(K)
sage: ff = f.change_ring(K)
sage: p = PP((c, 1))
sage: ff(ff(p)) == p
True
```python
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 + y^2, x*y])
sage: f.dynatomic_polynomial([2, 2])
x^4 + 4*x^2*y^2 + y^4
sage: R.<X> = PolynomialRing(QQ)
sage: K.<c> = NumberField(X^4 + 4*X^2 + 1)
sage: PP = P.change_ring(K)
sage: ff = f.change_ring(K)
sage: p = PP((c, 1))
sage: ff.nth_iterate(p, 4) == ff.nth_iterate(p, 2)
True
sage: P.<x,y> = ProjectiveSpace(CC, 1)
sage: f = DynamicalSystem_projective([x^2 - CC.0/3*y^2, y^2])
sage: f.dynatomic_polynomial(2)
(x^4*y + (-0.666666666666667*I)*x^2*y^3 - x*y^4 + (-0.111111111111111 - 0.333333333333333*I)*y^5)/(x^2*y - x*y^2 + (-0.333333333333333*I)*y^3)
```
```python
sage: R.<c> = QQ[]
sage: P.<x,y> = ProjectiveSpace(ZZ,1)
sage: f = DynamicalSystem_projective([x^2 + y^2, (1)*y^2 + (1)*x*y])
sage: f.dynatomic_polynomial([1,2]).parent()
Multivariate Polynomial Ring in x, y over Integer Ring

sage: P.<x, y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2 + y^2, y^2])
sage: f.dynatomic_polynomial(0)
0
sage: f.dynatomic_polynomial([0,0])
0
sage: f.dynatomic_polynomial(-1)
Traceback (most recent call last):
...  
TypeError: period must be a positive integer
```

Some rings still return SymbolicRing elements:

```python
sage: S.<t> = FunctionField(CC)
sage: P.<x,y> = ProjectiveSpace(S,1)
sage: f = DynamicalSystem_projective([t*x^2-1*y^2, t*y^2])
sage: f.dynatomic_polynomial([1, 2]).parent()  
Symbolic Ring
```

```python
sage: R.<x,y> = PolynomialRing(QQ)
sage: S = R.quo(R.ideal(y^2-x+1))
sage: P.<u,v> = ProjectiveSpace(FractionField(S),1)
sage: f = DynamicalSystem_projective([u^2 + S(x^2)*v^2, v^2])
sage: dyn = f.dynatomic_polynomial([1,1]); dyn
v^3*xbar^2 + u^2*v + u*v^2
sage: dyn.parent()  
Symbolic Ring
```

**green_function** (*P*, *v*, **kwds**)
Evaluate the local Green’s function at the place *v* for *P* with *N* terms of the series or to within a given error bound.

Must be over a number field or order of a number field. Note that this is the absolute local Green’s function so is scaled by the degree of the base field.

Use *v=0* for the archimedean place over *Q* or field embedding. Non-archimedean places are prime ideals for number fields or primes over *Q*.

**ALGORITHM:**
See Exercise 5.29 and Figure 5.6 of [Sil2007].

**INPUT:**
- *P* – a projective point
• \( v \) – non-negative integer. a place, use 0 for the archimedean place

kwds:
• \( N \) – (optional - default: 10) positive integer. number of terms of the series to use
• \( \text{prec} \) – (default: 100) positive integer, float point or \( p \)-adic precision
• \( \text{error_bound} \) – (optional) a positive real number

OUTPUT: a real number

EXAMPLES:

```python
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2+y^2, x*y]);
sage: Q = P(5, 1)
sage: f.green_function(Q, 0, N=30)
1.6460930159932946233759277576
```

```python
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2+y^2, x*y]);
sage: Q = P(5, 1)
sage: f.green_function(Q, 0, N=200, prec=200)
1.6460930160038721802875250367738355497198064992657997569827
```

```python
sage: K.<w> = QuadraticField(3)
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([17*x^2+1/7*y^2, 17*w*x*y])
sage: f.green_function(P.point([w, 2], False), K.places()[1])
1.7236334013785676107373093775
```

```python
sage: f.green_function(P([2, 1]), K.ideal(7), N=7)
0.48647753726382832627633818586
```

```python
sage: f.green_function(P([w, 1]), K.ideal(17), error_bound=0.001)
1.7236334013785676107373093775
```

```python
sage: f.green_function(P.point([5,2], False), 0, N=30)
1.7315451844777407992085512000
```

```python
sage: f.green_function(P.point([2,1], False), 0, N=30)
0.86577259223181088325226209526
```

```python
sage: f.green_function(P.point([1,1], False), 0, N=30)
0.43288629610862338612700146098
```

**height_difference_bound** \((\text{prec}=\text{None})\)

Return an upper bound on the different between the canonical height of a point with respect to this dynamical system and the absolute height of the point.

This map must be a morphism.

ALGORITHM:

Uses a Nullstellensatz argument to compute the constant. For details: see [Hutz2015].

INPUT:
• \( \text{prec} \) – (default: \text{RealField default}) positive integer, float point precision

OUTPUT: a real number

EXAMPLES:
```
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2+y^2, x*y])
sage: f.height_difference_bound()
1.38629436111989
```

This function does not automatically normalize.

```
sage: P.<x,y,z> = ProjectiveSpace(ZZ,2)
sage: f = DynamicalSystem_projective([4*x^2+100*y^2, 210*x*y, 10000*z^2])
sage: f.height_difference_bound()
11.0020998412042
sage: f.normalize_coordinates()
sage: f.height_difference_bound()
10.3089526606443
```

A number field example:

```
sage: R.<x> = QQ[]
sage: K.<c> = NumberField(x^3 - 2)
sage: P.<x,y,z> = ProjectiveSpace(K,2)
sage: f = DynamicalSystem_projective([1/(c+1)*x^2+c*y^2, 210*x*y, 10000*z^2])
sage: f.height_difference_bound()
11.002098412042

::

    sage: P.<x,y,z> = ProjectiveSpace(QQbar,2)
    sage: f = DynamicalSystem_projective([x^2, QQbar(sqrt(-1))*y^2, QQbar(sqrt(3))*z^2])
    sage: f.height_difference_bound()
    3.43967790223022
```

`is_PGL_minimal(prime_list=None)`

Check if this dynamical system is a minimal model in its conjugacy class.


INPUT:

* prime_list – (optional) list of primes to check minimality

OUTPUT: boolean

EXAMPLES:

```
sage: PS.<X,Y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([X^2+3*Y^2, X*Y])
sage: f.is_PGL_minimal()
True

sage: PS.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([6*x^2+12*x*y+7*y^2, 12*x*y])
sage: f.is_PGL_minimal()
False
```
is_postcritically_finite (err=0.01, embedding=None)

Determine if this dynamical system is post-critically finite.

Only for endomorphisms of \( \mathbb{P}^1 \). It checks if each critical point is preperiodic. The optional parameter \( \text{err} \) is passed into \( \text{is_preperiodic() as part of the preperiodic check.} \)

INPUT:

- \( \text{err} \) – (default: 0.01) positive real number
- \( \text{embedding} \) – embedding of base ring into \( \mathbb{Q} \)

OUTPUT: boolean

EXAMPLES:

```sage
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 - y^2, y^2])
sage: f.is_postcritically_finite()
True
```

```sage
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^3- y^3, y^3])
sage: f.is_postcritically_finite()
False
```

```sage
sage: R.<z> = QQ[]
sage: K.<v> = NumberField(z^8 + 3*z^6 + 3*z^4 + z^2 + 1)
sage: PS.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x^3+v*y^3, y^3])
sage: f.is_postcritically_finite(embedding=K.embeddings(QQbar)[0]) # long time
True
```

minimal_model (return_transformation=False, prime_list=None, algorithm=None)

Determine if this dynamical system is minimal.

This dynamical system must be defined over the projective line over the rationals. In particular, determine if this map is affine minimal, which is enough to decide if it is minimal or not. See Proposition 2.10 in [BM2012].

INPUT:

- \( \text{return_transformation} \) – (default: False) boolean; this signals a return of the \( PGL_2 \) transformation to conjugate this map to the calculated minimal model
- \( \text{prime_list} \) – (optional) a list of primes, in case one only wants to determine minimality at those specific primes
- \( \text{algorithm} \) – (optional) string; can be one of the following:
- 'BM' - the Bruin-Molnar algorithm [BM2012]
- 'HS' - the Hutz-Stoll algorithm [HS2018]

OUTPUT:

- a dynamical system on the projective line which is a minimal model of this map
- a $PGL(2, \mathbb{Q})$ element which conjugates this map to a minimal model

EXAMPLES:

```python
sage: PS.<X,Y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([X^2+3*Y^2, X*Y])
sage: f.minimal_model(return_transformation=True)
(Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (X : Y) to
  (X^2 + 3*Y^2 : X*Y),
  [1 0]
  [0 1])
```

```python
sage: PS.<X,Y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([7365/2*X^4 + 6282*X^3*Y + 4023*X^2*Y^2 + 1146*X*Y^3 + 245/2*Y^4,
  -12329/2 *X^4 - 10506*X^3*Y - 6723*X^2*Y^2 - 1914*X*Y^3 - 409/2*Y^4])
sage: f.minimal_model(return_transformation=True)
(Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (X : Y) to
  (9847*X^4 + 28088*X^3*Y + 30048*X^2*Y^2 + 14288*X*Y^3 + 2548*Y^4 : -12329*X^4 - 35164*X^3*Y - 37614*X^2*Y^2 - 17884*X*Y^3 - 3189*Y^4),
  [2 1]
  [0 1])
```

```python
sage: PS.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([6*x^2+12*x*y+7*y^2, 12*x*y])
sage: f.minimal_model()
Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to
  (x^2 + 12*x*y + 42*y^2 : 2*x*y)
```

```python
sage: PS.<x,y> = ProjectiveSpace(ZZ,1)
sage: f = DynamicalSystem_projective([6*x^2+12*x*y+7*y^2, 12*x*y + 42*y^2])
sage: g,M = f.minimal_model(return_transformation=True, algorithm='BM')
sage: f.conjugate(M) == g
True
```

```python
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem([2*x^2, y^2])
sage: f.minimal_model(return_transformation=True)
```

(continues on next page)
Dynamical System of Projective Space of dimension 1 over Rational Field
Defn: Defined on coordinates by sending (x : y) to
(x^2 : y^2),
[1 0]
[0 2]
sage: f.minimal_model(prime_list=[3])
Dynamical System of Projective Space of dimension 1 over Rational Field
Defn: Defined on coordinates by sending (x : y) to
(2*x^2 : y^2)
sage: PS.<X,Y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([X^2-Y^2, X^2+X*Y])
sage: f.minimal_model()
Traceback (most recent call last):
  ...TypeError: the function is not a morphism
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem([2*x^2, y^2])
sage: f.minimal_model(algorithm = 'BM')
Traceback (most recent call last):
  ...TypeError: affine minimality is only considered for maps not of the form f or
˓→1/f for a polynomial f

REFERENCES:
• [BM2012]
• [Mol2015]
• [HS2018]

multiplier \((P, n, check=True)\)

Return the multiplier of the point \(P\) of period \(n\) with respect to this dynamical system.

INPUT:

- \(P\) – a point on domain of this map
- \(n\) – a positive integer, the period of \(P\)
- \(check\) – (default: True) boolean; verify that \(P\) has period \(n\)

OUTPUT:

A square matrix of size \(\text{self.codomain().dimension_relative()}\) in the \(\text{base_ring}\) of this dynamical system.

EXAMPLES:

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: f = DynamicalSystem_projective([x^2,y^2, 4*z^2]);
sage: Q = P.point([4,4,1], False);
sage: f.multiplier(Q,1)
[2 0]
[0 2]
```
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([7*x^2 - 28*y^2, 24*x*y])
sage: f.multiplier(P(2,5), 4)
[231361/20736]

sage: P.<x,y> = ProjectiveSpace(CC,1)
sage: f = DynamicalSystem_projective([x^3 - 25*x*y^2 + 12*y^3, 12*y^3])
sage: f.multiplier(P(1,1), 5)
[0.389017489711934]

sage: P.<x,y> = ProjectiveSpace(RR,1)
sage: f = DynamicalSystem_projective([x^2-2*y^2, y^2])
sage: f.multiplier(P(2,1), 1)
[4.00000000000000]

sage: P.<x,y> = ProjectiveSpace(Qp(13),1)
sage: f = DynamicalSystem_projective([x^2-29/16*y^2, y^2])
sage: f.multiplier(P(5,4), 3)
[6 + 8*13 + 13^2 + 8*13^3 + 13^4 + 8*13^5 + 13^6 + 8*13^7 + 13^8 +
  8*13^9 + 13^10 + 8*13^11 + 13^12 + 8*13^13 + 13^14 + 8*13^15 + 13^16 +
  8*13^17 + 13^18 + 8*13^19 + O(13^20)]

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2-y^2, y^2])
sage: f.multiplier(P(0,1), 1)
Traceback (most recent call last):
  ... Value Error: (0 : 1) is not periodic of period 1

multiplier_spectra(n, formal=False, embedding=None, type='point')

Computes the \( n \) multiplier spectra of this dynamical system.

This is the set of multipliers of the periodic points of formal period \( n \) included with the appropriate multiplicity. User can also specify to compute the \( n \) multiplier spectra instead which includes the multipliers of all periodic points of period \( n \). The map must be defined over projective space of dimension 1 over a number field.

INPUT:

- \( n \) – a positive integer, the period
- formal – (default: False) boolean; True specifies to find the formal \( n \) multiplier spectra of this map and False specifies to find the \( n \) multiplier spectra
- embedding – embedding of the base field into \( \mathbb{Q} \)
- type – (default: 'point') string; either 'point' or 'cycle' depending on whether you compute one multiplier per point or one per cycle

OUTPUT: a list of \( \mathbb{Q} \) elements

EXAMPLES:

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([4608*x^10 - 2910096*x^9*y + 325988068*x^8*y^2 + 3182519832*x^7*y^3 - 4139806626613*x^6*y^4 - 4439736715486*x^5*y^5 + 2317935971590902*x^4*y^6 - 15344764859590852*x^3*y^7 + 2561851642765275*x^2*y^8\( 208 Chapter 6. Arithmetic Dynamical Systems

(continues on next page)
+ 113578270285012470*x*y^9 - 150049940203963800*y^10, 4608*y^10]
sage: f.multiplier_spectra(1)
[0, -7198147681176255644585/256, 848446157556848459363/19683, ...
\rightarrow 3323781962860268721722583135/35184372088832,
52978480109921/256, -4290991999449436653/2097152, 1061953534167447403/19683,
\rightarrow -3086380435599991/9,
82911372672808161930567/8192, -1198205023656808439999, ...
\rightarrow 3553497751559301575157261317/8192]
sage: setVerbose(None)
sage: z = QQ['z'].0
sage: K.<w> = NumberField(z^4 - 4*z^2 + 1,'z')
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x^2 - w/4*y^2, y^2])
sage: f.multiplier_spectra(2, formal=False, embedding=K.embeddings(QQbar)[0], ...
\rightarrow type='cycle')
[0, 5.931851652578137? + 0.?e-49*I,
0.0681483474218635? - 1.930649271699173?*I,
0.0681483474218635? + 1.930649271699173?*I]
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 - 3/4*y^2, y^2])
sage: f.multiplier_spectra(2, formal=False, type='cycle')
[0, 1, 1, 9]
sage: f.multiplier_spectra(2, formal=False, type='point')
[0, 1, 1, 1, 1, 9]
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 - 7/4*y^2, y^2])
sage: f.multiplier_spectra(3, formal=True, type='cycle')
[1, 1]
sage: f.multiplier_spectra(3, formal=True, type='point')
[1, 1, 1, 1, 1, 1]
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 + y^2, x*y])
sage: f.multiplier_spectra(1)
[1, 1, 1]

nth_iterate(P, n, **kwds)
Return the n-th iterate of the point P by this dynamical system.
If normalize is True, then the coordinates are automatically normalized.

Todo: Is there a more efficient way to do this?

Todo: Is there a more efficient way to do this?

Todo: Is there a more efficient way to do this?

Todo: Is there a more efficient way to do this?

Todo: Is there a more efficient way to do this?

Todo: Is there a more efficient way to do this?

Todo: Is there a more efficient way to do this?

Todo: Is there a more efficient way to do this?

Todo: Is there a more efficient way to do this?

Todo: Is there a more efficient way to do this?

Todo: Is there a more efficient way to do this?

Todo: Is there a more efficient way to do this?

Todo: Is there a more efficient way to do this?
OUTPUT: a point in this map’s codomain

EXAMPLES:

```python
sage: P.<x,y> = ProjectiveSpace(ZZ,1)
sage: f = DynamicalSystem_projective([x^2+y^2, 2*y^2])
sage: Q = P(1,1)
sage: f.nth_iterate(Q,4)
(32768 : 32768)
```

```python
sage: P.<x,y> = ProjectiveSpace(ZZ,1)
sage: f = DynamicalSystem_projective([x^2+y^2, 2*y^2])
sage: Q = P(1,1)
sage: f.nth_iterate(Q, 4, normalize=True)
(1 : 1)
```

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: f = DynamicalSystem_projective([x^2, 2*y^2, z^2-x^2])
sage: Q = P(2,7,1)
sage: f.nth_iterate(Q,2)
(-16/7 : -2744 : 1)
```

```python
sage: R.<t> = PolynomialRing(QQ)
sage: P.<x,y,z> = ProjectiveSpace(R,2)
sage: f = DynamicalSystem_projective([x^2+t*y^2, (2-t)*y^2, z^2])
sage: Q = P(2+t,7,t)
sage: f.nth_iterate(Q,2)
(t^4 + 2507*t^3 - 6787*t^2 + 10028*t + 16 : -2401*t^3 + 14406*t^2 - 28812*t + 19208 : t^4)
```

```python
sage: P.<x,y,z> = ProjectiveSpace(ZZ,2)
sage: X = P.subscheme(x^2-y^2)
sage: f = DynamicalSystem_projective([x^2, y^2, z^2], domain=X)
sage: f.nth_iterate(X(2,2,3), 3)
(256 : 256 : 6561)
```

```python
sage: K.<c> = FunctionField(QQ)
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x^3 - 2*x*y^2 - c*y^3, x*y^2])
sage: f.nth_iterate(P(c,1), 2)
((c^6 - 9*c^4 + 25*c^2 - c - 21)/(c^2 - 3) : 1)
```

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: H = Hom(P,P)
sage: f = H([x^2+3*y^2, 2*y^2, z^2])
sage: P(2, 7, 1).nth_iterate(f, -2)
Traceback (most recent call last):
...
TypeError: must be a forward orbit
```

```python
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^3, x*y^2], domain=P)
sage: f.nth_iterate(P(0, 1), 3, check=False)
(0 : 0)
sage: f.nth_iterate(P(0, 1), 3)
Traceback (most recent call last):
```
...  

```
ValueError: [0, 0] does not define a valid point since all entries are 0
```

```
sage: P.<x,y> = ProjectiveSpace(ZZ, 1)
sage: f = DynamicalSystem_projective([x^3, x*y^2], domain=P)
sage: f.nth_iterate(P(2,1), 3, normalize=False)
(134217728 : 524288)
sage: f.nth_iterate(P(2,1), 3, normalize=True)
(256 : 1)
```

**nth_iterate_map** \( n, \text{normalize=False} \)

Return the \( n \)-th iterate of this dynamical system.

**ALGORITHM:**

Uses a form of successive squaring to reducing computations.

**Todo:** This could be improved.

**INPUT:**

- \( n \) – positive integer
- \( \text{normalize} \) – boolean; remove gcd’s during iteration

**OUTPUT:** a projective dynamical system

**EXAMPLES:**

```
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2+y^2, y^2])
sage: f.nth_iterate_map(2)
Dynamical System of Projective Space of dimension 1 over Rational Field
Defn: Defined on coordinates by sending (x : y) to
(x^4 + 2*x^2*y^2 + 2*y^4 : y^4)
```

```
sage: P.<x,y> = ProjectiveSpace(CC,1)
sage: f = DynamicalSystem_projective([x^2-y^2, x*y])
sage: f.nth_iterate_map(3)
Dynamical System of Projective Space of dimension 1 over Complex Field with 53 bits of precision
Defn: Defined on coordinates by sending (x : y) to
(x^8 + (-7.00000000000000)*x^6*y^2 + 13.00000000000000*x^4*y^4 +
(-7.00000000000000)*x^2*y^6 + y^8 : x^7*y + (-4.00000000000000)*x^5*y^3 +
4.000000000000000*x^3*y^5 - x*y^7)
```

```
sage: P.<x,y,z> = ProjectiveSpace(ZZ,2)
sage: f = DynamicalSystem_projective([x^2-y^2, x*y, z^2+x^2])
sage: f.nth_iterate_map(2)
Dynamical System of Projective Space of dimension 2 over Integer Ring
Defn: Defined on coordinates by sending (x : y : z) to
(x^4 - 3*x^2*y^2 + y^4 : x^3*y - x*y^3 : 2*x^4 - 2*x^2*y^2 + y^4 +
2*x^2*z^2 + z^4)
```

6.3. Dynamical systems on projective schemes
sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: X = P.subscheme(x*z-y^2)
sage: f = DynamicalSystem_projective([x^2, x*z, z^2], domain=X)
sage: f.nth_iterate_map(2)
Dynamical System of Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
-y^2 + x*z
Defn: Defined on coordinates by sending (x : y : z) to
(x^4 : x^2*z^2 : z^4)

sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([y^2*z^3, y^3*z^2, x^5])
sage: f.nth_iterate_map( 5, normalize=True)
Dynamical System of Projective Space of dimension 2 over Rational Field
Defn: Defined on coordinates by sending (x : y : z) to
(y^202*z^443 : x^140*y^163*z^342 : x^645)

orbit \((P, N, **\text{kwds})\)

Return the orbit of the point \(P\) by this dynamical system.

Let \(F\) be this dynamical system. If \(N\) is an integer return \([P, F(P), \ldots, F^N(P)]\). If \(N\) is a list or tuple \(N = [m, k]\) return \([F^m(P), \ldots, F^k(P)]\). Automatically normalize the points if \(\text{normalize}=\text{True}\). Perform the checks on point initialization if \(\text{check}=\text{True}\).

INPUT:

- \(P\) – a point in this dynamical system’s domain
- \(n\) – a non-negative integer or list or tuple of two non-negative integers

kwds:

- \(\text{check}=(\text{default: True})\) boolean
- \(\text{normalize}=(\text{default: False})\) boolean

OUTPUT: a list of points in this dynamical system’s codomain

EXAMPLES:

sage: P.<x,y,z> = ProjectiveSpace(ZZ,2)
sage: f = DynamicalSystem_projective([x^2+y^2, y^2-z^2, 2*z^2])
sage: f.orbit(P(1,2,1), 3)
[(1 : 2 : 1), (5 : 3 : 2), (34 : 5 : 8), (1181 : -39 : 128)]

sage: P.<x,y,z> = ProjectiveSpace(ZZ,2)
sage: f = DynamicalSystem_projective([x^2+y^2, y^2-z^2, 2*z^2])
sage: f.orbit(P(1,2,1), [2,4])
[(34 : 5 : 8), (1181 : -39 : 128), (1396282 : -14863 : 32768)]

sage: P.<x,y,z> = ProjectiveSpace(ZZ,2)
sage: X = P.subscheme(x^2-y^2)
sage: f = DynamicalSystem_projective([x^2, y^2, x*z], domain=X)
sage: f.orbit(X(2,2,3), 3, normalize=True)
[(2 : 2 : 3), (2 : 2 : 3), (2 : 2 : 3), (2 : 2 : 3)]

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2+y^2, y^2])
(continues on next page)
sage: K.<c> = FunctionField(QQ)
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x^2+c*y^2, y^2])
sage: f.orbit(P(0,1), 3)
[(0 : 1), (c : 1), (c^2 + c : 1), (c^4 + 2*c^3 + c^2 + c : 1)]

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2+y^2, y^2], domain=P)
sage: f.orbit(P.point([1, 2], False), 4, check=False)
[(1 : 2), (5 : 4), (41 : 16), (1937 : 256), (3817505 : 65536)]

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2, 2*y^2], domain=P)
sage: P(2, 1).orbit(f,[-1, 4])
Traceback (most recent call last):
  ... 
TypeError: orbit bounds must be non-negative
sage: P(2, 1).orbit(f, 0.1)
Traceback (most recent call last):
  ... 
TypeError: Attempt to coerce non-integral RealNumber to Integer

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^3, x*y^2], domain=P)
sage: P(0, 1).orbit(f, 3)
Traceback (most recent call last):
  ... 
ValueError: [0, 0] does not define a valid point since all entries are 0
sage: P(0, 1).orbit(f, 3, check=False)
[(0 : 1), (0 : 0), (0 : 0), (0 : 0)]

sage: P.<x,y> = ProjectiveSpace(ZZ, 1)
sage: f = DynamicalSystem_projective([x^3, x*y^2], domain=P)
sage: P(2,1).orbit(f, 3, normalize=False)
[(2 : 1), (8 : 2), (512 : 32), (134217728 : 524288)]
sage: P(2, 1).orbit(f, 3, normalize=True)
[(2 : 1), (4 : 1), (16 : 1), (256 : 1)]

The periodic points of a dynamical system can be computed using the `periodic_points` method.

```python
periodic_points(n, minimal=True, R=None, algorithm='variety', return_scheme=False)
```

This method computes the periodic points of period `n` for a given dynamical system defined over a ring `R` or the base ring of the map.

For rational maps, this method can be used to find periodic points of a given period. Note that this method will include the indeterminacy locus.

**INPUT:**

1. `n`: The period of the points to be computed.
2. `minimal`: A boolean indicating whether to find minimal periodic points. Default is `True`.
3. `R`: The ring over which the dynamical system is defined. Default is `None`, which means the base ring of the map.
4. `algorithm`: A string indicating the algorithm to use. Default is `'variety'`.
5. `return_scheme`: A boolean indicating whether to return the scheme. Default is `False`.

This method is particularly useful for studying the dynamics of systems defined over finite fields or rings.
• \( n \) - a positive integer
• \text{minimal} – (default: True) boolean; True specifies to find only the periodic points of minimal period \( n \) and False specifies to find all periodic points of period \( n \)
• \( R \) a commutative ring
• \text{algorithm} – (default: 'variety') must be one of the following:
  – 'variety' - find the rational points on the appropriate variety
  – 'cyclegraph' - find the cycles from the cycle graph
• \text{return\_scheme} – return a subscheme of the ambient space that defines the \( n \) th periodic points

\textbf{OUTPUT:}

A list of periodic points of this map or the subscheme defining the periodic points.

\textbf{EXAMPLES:}

```python
sage: set_verbose(None)
sage: P.<x,y> = ProjectiveSpace(QQbar,1)
sage: f = DynamicalSystem_projective([(x^2-xy+y^2, x^2-y^2+x*y)])
sage: f.periodic_points(1)
[(-0.500000000000000? - 0.866025403784439?*I : 1),
 (-0.500000000000000? + 0.866025403784439?*I : 1),
 (1 : 1)]
```

```python
sage: P.<x,y,z> = ProjectiveSpace(QQuadraticField(5,'t'),2)
sage: f = DynamicalSystem_projective([(x^2 - 21/16*z^2, y^2-z^2, z^2)])
sage: f.periodic_points(2)
[(-5/4 : -1 : 1), (-5/4 : -1/2*t + 1/2 : 1), (-5/4 : 0 : 1),
 (-5/4 : 1/2*t + 1/2 : 1), (-3/4 : -1 : 1), (-3/4 : 0 : 1),
 (1/4 : -1 : 1), (1/4 : -1/2*t + 1/2 : 1), (1/4 : 0 : 1),
 (1/4 : 1/2*t + 1/2 : 1), (7/4 : -1 : 1), (7/4 : 0 : 1)]
```

```python
sage: w = QQ['w'].0
sage: K = NumberField(w^6 - 3*w^5 + 5*w^4 - 5*w^3 + 5*w^2 - 3*w + 1,'s')
sage: P.<x,y,z> = ProjectiveSpace(K,2)
sage: f = DynamicalSystem_projective([(x^2+z^2, y^2+x^2, z^2+y^2)])
sage: f.periodic_points(1)
[(-s^5 + 3*s^4 - 5*s^3 + 4*s^2 - 3*s + 1 : s^5 - 2*s^4 + 3*s^3 - 3*s^2 + 4*s -
 \rightarrow 1 : 1),
 (-2*s^5 + 4*s^4 - 5*s^3 + 3*s^2 - 4*s : -2*s^5 + 5*s^4 - 7*s^3 + 6*s^2 - 7*s_\rightarrow + 3 : 1),
 (-s^5 + 3*s^4 - 4*s^3 + 4*s^2 - 4*s + 2 : -s^5 + 2*s^4 - 2*s^3 + s^2 - s_\rightarrow \rightarrow + 2 : 1),
 (s^5 - 2*s^4 + 3*s^3 - 3*s^2 + 3*s - 1 : -s^5 + 3*s^4 - 5*s^3 + 4*s^2 - 4*s_\rightarrow + 2 : 1),
 (2*s^5 - 6*s^4 + 9*s^3 - 8*s^2 + 7*s - 4 : 2*s^5 - 5*s^4 + 7*s^3 - 5*s^2 + 4*s^\rightarrow + 6 : 2 : 1),
 (1 : 1 : 1),
 (s^5 - 2*s^4 + 2*s^3 + s : s^5 - 3*s^4 + 4*s^3 - 3*s^2 + 9*s - 1 : 1)]
```

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: f = DynamicalSystem_projective([(x^2 - 21/16*z^2, y^2-2*z^2, z^2)])
sage: f.periodic_points(2, False)
 (1 : 1 : 1),
 (-5/4 : -1/2*t + 1/2 : 1), (-3/4 : -1/2*t + 1/2 : 1), (1/4 : -1 : 1),
 (1/4 : 0 : 1),
 (1/4 : 2 : 1),
 (7/4 : -1 : 1), (7/4 : 0 : 1),
 (7/4 : 2 : 1),
 (1 : 1 : 1),
 (s^5 - 2*s^4 + 2*s^3 + s : s^5 - 3*s^4 + 4*s^3 - 3*s^2 + 9*s - 1 : 1)]
```

(continues on next page)
\((-3/4 : 2 : 1), (0 : 1 : 0), (1/4 : -1 : 1), (1/4 : 2 : 1), (1 : 0 : 0), (1 : 1 : 0), (7/4 : -1 : 1), (7/4 : 2 : 1)\]

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: f = DynamicalSystem_projective([x^2 - 21/16*z^2, y^2-2*z^2, z^2])
sage: f.periodic_points(2)
[(-5/4 : -1 : 1), (-5/4 : 2 : 1), (1/4 : -1 : 1), (1/4 : 2 : 1)]
```

```python
sage: set_verbose(None)
sage: P.<x,y> = ProjectiveSpace(ZZ, 1)
sage: f = DynamicalSystem_projective([x^2+y^2,y^2])
sage: f.periodic_points(2, R=QQbar, minimal=False)
[(-0.500000000000000? - 1.322875655532296?*I : 1),
 (-0.500000000000000? + 1.322875655532296?*I : 1),
 (0.500000000000000? - 0.866025403784439?*I : 1),
 (0.500000000000000? + 0.866025403784439?*I : 1),
 (1 : 0)]
```

```python
sage: P.<x,y> = ProjectiveSpace(GF(307), 1)
sage: f = DynamicalSystem_projective([x^10+y^10, y^10])
sage: f.periodic_points(16, minimal=True, algorithm='cyclegraph')
[(69 : 1), (185 : 1), (120 : 1), (136 : 1), (97 : 1), (183 : 1),
 (170 : 1), (105 : 1), (274 : 1), (275 : 1), (154 : 1), (156 : 1),
 (87 : 1), (95 : 1), (161 : 1), (128 : 1)]
```

```python
sage: P.<x,y> = ProjectiveSpace(GF(13^2,'t'),1)
sage: f = DynamicalSystem_projective([x^3 + 3*y^3, x^2*y])
sage: f.periodic_points(30, minimal=True, algorithm='cyclegraph')
[(t + 3 : 1), (6*t + 6 : 1), (7*t + 1 : 1), (2*t + 8 : 1),
 (3*t + 4 : 1), (10*t + 12 : 1), (8*t + 10 : 1), (5*t + 11 : 1),
 (7*t + 4 : 1), (4*t + 8 : 1), (9*t + 1 : 1), (2*t + 2 : 1),
 (11*t + 9 : 1), (5*t + 7 : 1), (t + 10 : 1), (12*t + 4 : 1),
 (7*t + 12 : 1), (6*t + 8 : 1), (11*t + 10 : 1), (10*t + 7 : 1),
 (3*t + 9 : 1), (5*t + 5 : 1), (8*t + 3 : 1), (6*t + 11 : 1),
 (9*t + 12 : 1), (4*t + 10 : 1), (11*t + 4 : 1), (2*t + 7 : 1),
 (8*t + 12 : 1), (12*t + 11 : 1)]
```

```python
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([3*x^2+5*y^2,y^2])
sage: f.periodic_points(2, R=GF(3), minimal=False)
[(2 : 1)]
```

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([x^2, x*y, z^2])
sage: f.periodic_points(1)
Traceback (most recent call last):
...  
  TypeError: use return_scheme=True
```

```python
sage: R.<x> = QQ[]
sage: K.<u> = NumberField(x^2 - x + 3)
sage: P.<x,y,z> = ProjectiveSpace(K,2)
sage: X = P.subscheme(2*x-y)
sage: f = DynamicalSystem_projective([x^2-y^2, 2*(x^2-y^2), y^2-z^2],
  domain=X)
```

6.3. Dynamical systems on projective schemes
\begin{lstlisting}[language=sage]
sage: f.periodic_points(2)
[(-1/5*u - 1/5 : -2/5*u - 2/5 : 1), (1/5*u - 2/5 : 2/5*u - 4/5 : 1)]

sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: f = DynamicalSystem_projective([x^2-y^2, x^2-z^2, y^2-z^2])
sage: f.periodic_points(1)
[(-1 : 0 : 1)]
sage: f.periodic_points(1, return_scheme=True)
Closed subscheme of Projective Space of dimension 2 over Rational Field
defined by:
-x^3 + x^2*y - y^3 + x*z^2,
-x*y^2 + x^2*z - y^2*z + x*z^2,
-y^3 + x^2*z + y*z^2 - z^3
sage: f.periodic_points(2, minimal=True, return_scheme=True)
Traceback (most recent call last):
... Not ImplementedError: return_subscheme only implemented for minimal=False

\end{lstlisting}

possible_periods(**kwds)

Return the set of possible periods for rational periodic points of this dynamical system.

Must be defined over \( \mathbb{Z} \) or \( \mathbb{Q} \).

**ALGORITHM:**

Calls \texttt{self.possible_periods()} modulo all primes of good reduction in range \texttt{prime_bound}.

Return the intersection of those lists.

**INPUT:**

\texttt{kwds}:

\begin{itemize}
  \item \texttt{prime_bound} – (default: \[1, 20\]) a list or tuple of two positive integers or an integer for the upper bound
  \item \texttt{bad_primes} – (optional) a list or tuple of integer primes, the primes of bad reduction
  \item \texttt{ncpus} – (default: all cpus) number of cpus to use in parallel
\end{itemize}

**OUTPUT:** a list of positive integers

**EXAMPLES:**

\begin{lstlisting}[language=sage]
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2-29/16*y^2, y^2])
sage: f.possible_periods(ncpus=1)
[1, 3]
sage: PS.<x,y> = ProjectiveSpace(1,QQ)
sage: f = DynamicalSystem_projective([5*x^3 - 53*x*y^2 + 24*y^3, 24*y^3])
sage: f.possible_periods(prime_bound=[1,5])
Traceback (most recent call last):
... ValueError: no primes of good reduction in that range
sage: f.possible_periods(prime_bound=[1,10])
[1, 4, 12]
sage: f.possible_periods(prime_bound=[1,20])
[1, 4]
\end{lstlisting}

\end{itemize}
\begin{verbatim}
 sage: P.<x,y,z> = ProjectiveSpace(ZZ,2)
 sage: f = DynamicalSystem_projective([2*x^3 - 50*x*z^2 + 24*z^3, ....: 5*y^3 - 53*y*z^2 + 24*z^3, 24*z^3])
 sage: f.possible_periods(prime_bound=10)
 [1, 2, 6, 20, 42, 60, 140, 420]
 sage: f.possible_periods(prime_bound=20)
 # long time
 [1, 20]

primes_of_bad_reduction (check=True)
Determine the primes of bad reduction for this dynamical system.

Must be defined over a number field.

If check is True, each prime is verified to be of bad reduction.

ALGORITHM:

\(p\) is a prime of bad reduction if and only if the defining polynomials of self have a common zero. Or stated another way, \(p\) is a prime of bad reduction if and only if the radical of the ideal defined by the defining polynomials of self is not \((x_0, x_1, \ldots, x_N)\). This happens if and only if some power of each \(x_i\) is not in the ideal defined by the defining polynomials of self. This last condition is what is checked. The lcm of the coefficients of the monomials \(x_i\) in a Groebner basis is computed. This may return extra primes.

INPUT:

- \(\text{check}\) – (default: True) boolean

OUTPUT: a list of primes

EXAMPLES:

\begin{verbatim}
 sage: P.<x,y> = ProjectiveSpace(QQ,1)
 sage: f = DynamicalSystem_projective([1/3*x^2+1/2*y^2, y^2])
 sage: f.primes_of_bad_reduction()
 [2, 3]

 sage: P.<x,y,z,w> = ProjectiveSpace(QQ,3)
 sage: f = DynamicalSystem_projective([12*x*z-7*y^2, 31*x^2-y^2, 26*z^2, 3*w^2-z*w])
 sage: f.primes_of_bad_reduction()
 [2, 3, 7, 13, 31]

 A number field example:

\begin{verbatim}
 sage: R.<z> = QQ[]
 sage: K.<a> = NumberField(z^2 - 2)
 sage: P.<x,y> = ProjectiveSpace(K,1)
 sage: f = DynamicalSystem_projective([1/3*x^2+1/a*y^2, y^2])
 sage: f.primes_of_bad_reduction()
 [Fractional ideal (a), Fractional ideal (3)]
\end{verbatim}

This is an example where check = False returns extra primes:

\begin{verbatim}
 sage: P.<x,y,z> = ProjectiveSpace(ZZ,2)
 sage: f = DynamicalSystem_projective([3*x*y^2 + 7*y^3 - 4*y^2*z + 5*z^3, ....: -5*x^3 + x^2*y + y^3 + 2*x^2*z, ....: -2*x^2*y + x*y^2 + y^3 - 4*y^2*z + x*z^2])
 sage: f.primes_of_bad_reduction(False)
\end{verbatim}
\end{verbatim}
\end{verbatim}

6.3. Dynamical systems on projective schemes 217
reduced_form(**kwds)

Return reduced form of this dynamical system.

The reduced form is the $SL(2, \mathbb{Z})$ equivalent morphism obtained by applying the binary form reduction algorithm from Stoll and Cremona [CS2003] to the homogeneous polynomial defining the periodic points (the dynatomic polynomial). The smallest period $n$ with enough periodic points is used and without roots of too large multiplicity.

This should also minimize the size of the coefficients, but this is not always the case. By default the coefficient minimizing algorithm in [HS2018] is applied.

See `sage.rings.polynomial.multi_polynomial.reduced_form()` for the information on binary form reduction.

Implemented by Rebecca Lauren Miller as part of GSOC 2016. Minimal height added by Ben Hutz July 2018.

**INPUT:**

keywords:

- `prec` – (default: 300) integer, desired precision
- `return_conjugation` – (default: True) boolean; return an element of $SL(2, \mathbb{Z})$
- `error_limit` – (default: 0.000001) a real number, sets the error tolerance
- `smallest_coeffs` – (default: True), boolean, whether to find the model with smallest coefficients
- `dynatomic` – (default: True) boolean, to use formal periodic points
- `start_n` – (default: 1), positive integer, firs period to rry to find appropriate binary form
- `emb` – (optional) embedding of based field into CC
- `algorithm` – (optional) which algorithm to use to find all minimal models. Can be one of the following:
  - 'BM' – Bruin-Molnar algorithm [BM2012]
  - 'HS' – Hutz-Stoll algorithm [HS2018]
- `check_minimal` – (default: True), boolean, whether to check if this map is a minimal model
- `smallest_coeffs` – (default: True), boolean, whether to find the model with smallest coefficients

**OUTPUT:**

- a projective morphism
- a matrix

**EXAMPLES:**

```python
sage: PS.<x,y> = ProjectiveSpace(QQ, 1)
(sage: f = DynamicalSystem_projective([x^3 + x*y^2, y^3])
(sage: m = matrix(QQ, 2, 2, [-201221, -1, 1, 0])
(sage: f = f.conjugate(m)
(sage: f.reduced_form(prec=50, smallest_coeffs=False) #needs 2 periodic
```

```python
Traceback (most recent call last):
```
...  
ValueError: accuracy of Newton's root not within tolerance(0.000066... > 1e-06), increase precision  
sage: f.reduced_form(smallest_coeffs=False)  
(  
Dynamical System of Projective Space of dimension 1 over Rational Field  
  Defn: Defined on coordinates by sending (x : y) to  
(x^3 + x*y^2 : y^3)  
,  
[ 0 -1]  
[ 1 201221]  
)  
sage: PS.<x,y> = ProjectiveSpace(ZZ, 1)  
sage: f = DynamicalSystem_projective([x^2+ x*y, y^2]) #needs 3 periodic  
sage: m = matrix(QQ, 2, 2, [-221, -1, 1, 0])  
sage: f = f.conjugate(m)  
sage: f.reduced_form(prec=200, smallest_coeffs=False)  
(  
Dynamical System of Projective Space of dimension 1 over Integer Ring  
  Defn: Defined on coordinates by sending (x : y) to  
(-x^2 + x*y - y^2 : -y^2)  
,  
[ 0 -1]  
[ 1 220]  
)  
sage: P.<x,y> = ProjectiveSpace(QQ, 1)  
sage: f = DynamicalSystem_projective([x^3, y^3])  
sage: f.reduced_form(smallest_coeffs=False)  
(  
Dynamical System of Projective Space of dimension 1 over Rational Field  
  Defn: Defined on coordinates by sending (x : y) to  
(x^3 : y^3)  
,  
[1 0]  
[0 1]  
)  
sage: PS.<X,Y> = ProjectiveSpace(QQ,1)  
sage: f = DynamicalSystem_projective([7365*X^4 + 12564*X^3*Y + 8046*X^2*Y^2 + ...
-2282*X*Y^3 + 245*Y^4, 
-12329*X^4 - 21012*X^3*Y - 13446*X^2*Y^2 - 3828*X*Y^3 - 409*Y^4])  
sage: f.reduced_form(prec=30, smallest_coeffs=False)  
Traceback (most recent call last):  
...  
ValueError: accuracy of Newton's root not within tolerance(0.00008... > 1e-06), increase precision  
sage: f.reduced_form(smallest_coeffs=False)  
(  
Dynamical System of Projective Space of dimension 1 over Rational Field  
  Defn: Defined on coordinates by sending (X : Y) to  
(-7*X^4 - 12*X^3*Y - 42*X^2*Y^2 - 12*X*Y^3 - 7*Y^4 : -X^4 - 4*X^3*Y - ...
-6*X^2*Y^2 - 4*X*Y^3 - Y^4),
(... continues on next page)
[-1  2]
[ 2 -5]
)

\texttt{sage}: \texttt{P.<x,y> = ProjectiveSpace(RR, 1)}
\texttt{sage}: \texttt{f = DynamicalSystem_projective([x^4, RR(sqrt(2))*y^4])}
\texttt{sage}: \texttt{m = matrix(RR, 2, 2, [1,12,0,1])}
\texttt{sage}: \texttt{f = f.conjugate(m)}
\texttt{sage}: \texttt{f.reduced_form(smallest_coeffs=False)}

\texttt{(Dynamical System of Projective Space of dimension 1 over Real Field with 53 \rightarrow \text{bits of precision}}
\texttt{Defn: Defined on coordinates by sending (x : y) to}
\texttt{(x^4 - 2.86348722511320e-12*y^4 : 1.41421356237310*y^4)}
,
[ 1 -12]
[ 0  1]
)

\texttt{sage}: \texttt{P.<x,y> = ProjectiveSpace(CC, 1)}
\texttt{sage}: \texttt{f = DynamicalSystem_projective([x^4, CC(sqrt(-2))*y^4])}
\texttt{sage}: \texttt{m = matrix(CC, 2, 2, [1,12,0,1])}
\texttt{sage}: \texttt{f = f.conjugate(m)}
\texttt{sage}: \texttt{f.reduced_form(smallest_coeffs=False)}

\texttt{(Dynamical System of Projective Space of dimension 1 over Complex Field with 53 \rightarrow \text{bits of precision}}
\texttt{Defn: Defined on coordinates by sending (x : y) to}
\texttt{(x^4 + (1.03914726748259e-15)*y^4 : (8.65956056235493e-17 + 1.41421356237309*I)*y^4)}
,
[ 1 -12]
[ 0  1]
)

\texttt{sage}: \texttt{K.<w> = QuadraticField(2)}
\texttt{sage}: \texttt{P.<x,y> = ProjectiveSpace(K, 1)}
\texttt{sage}: \texttt{f = DynamicalSystem_projective([x^3, w*y^3])}
\texttt{sage}: \texttt{m = matrix(K, 2, 2, [1,12,0,1])}
\texttt{sage}: \texttt{f = f.conjugate(m)}
\texttt{sage}: \texttt{f.reduced_form(smallest_coeffs=False)}

\texttt{(Dynamical System of Projective Space of dimension 1 over Number Field in w with defining polynomial x^2 - 2}
\texttt{Defn: Defined on coordinates by sending (x : y) to}
\texttt{(x^3 : (w)*y^3)}
,
[ 1 -12]
[ 0  1]
)
sage: R.<x> = QQ[]
sage: K.<w> = NumberField(x^5+x-3, embedding=(x^5+x-3).roots(ring=CC)[0][0])
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: f = DynamicalSystem_projective([12*x^3, 2334*w*y^3])
sage: m = matrix(K, 2, 2, [-12,1,1,0])
sage: f = f.conjugate(m)
sage: f.reduced_form(smallest_coeffs=False)

Dynamical System of Projective Space of dimension 1 over Number Field
in w with defining polynomial x^5 + x - 3
Defn: Defined on coordinates by sending (x : y) to
(12*x^3 : (2334*w)*y^3)

sage: P.<x,y> = QQ[]
sage: f = DynamicalSystem([-4*y^2, 9*x^2 - 12*x*y])
sage: f.reduced_form()

Dynamical System of Projective Space of dimension 1 over Rational Field
Defn: Defined on coordinates by sending (x : y) to
(-2*x^2 + 2*y^2 : x^2 + 2*y^2)

sage: P.<x,y> = QQ[]
sage: f = DynamicalSystem([-2*x^3 - 9*x^2*y - 12*x*y^2 - 6*y^3 , y^3])
sage: f.reduced_form()

Dynamical System of Projective Space of dimension 1 over Rational Field
Defn: Defined on coordinates by sending (x : y) to
(x^3 + 3*x^2*y : 3*x*y^2 + y^3)

sage: P.<x,y> = QQ[]
sage: f = DynamicalSystem([4*x^2 - 7*y^2, 4*y^2])
sage: f.reduced_form(start_n=2, dynatomic=False) #long time

Dynamical System of Projective Space of dimension 1 over Rational Field
Defn: Defined on coordinates by sending (x : y) to
(x^2 - x*y - y^2 : y^2)

6.3. Dynamical systems on projective schemes 221
sage: P.<x,y> = QQ[]
sage: f = DynamicalSystem([4*x^2 + y^2, 4*y^2])
sage: f.reduced_form()  # long time
(Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to
  (x^2 - x*y + y^2 : y^2),
  [ 2 -1]
  [ 0  2])

resultant (normalize=False)
Computes the resultant of the defining polynomials of this dynamical system.

If normalize is True, then first normalize the coordinate functions with
normalize_coordinates().

INPUT:

• normalize – (default: False) boolean

OUTPUT: an element of the base ring of this map

EXAMPLES:

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2+y^2, 6*y^2])
sage: f.resultant()
36

sage: R.<t> = PolynomialRing(GF(17))
sage: P.<x,y> = ProjectiveSpace(R,1)
sage: f = DynamicalSystem_projective([t*x^2+t*y^2, 6*y^2])
sage: f.resultant()
2*t^2

sage: R.<t> = PolynomialRing(GF(17))
sage: P.<x,y,z> = ProjectiveSpace(R,2)
sage: f = DynamicalSystem_projective([t*x^2+t*y^2, 6*y^2, 2*t*z^2])
sage: f.resultant()
13*t^8

sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: F = DynamicalSystem_projective([x^2+y^2,6*y^2,10*x*z+z^2+y^2])
sage: F.resultant()
1296

sage: R.<t>=PolynomialRing(QQ)
sage: s = (t^3+t+1).roots(QQbar)[0][0]
sage: P.<x,y>=ProjectiveSpace(QQbar,1)
sage: f = DynamicalSystem_projective([[s*x^3-13*y^3, y^3-15*y^3]])
sage: f.resultant()
871.6925062959149?

sigma_invariants (n, formal=False, embedding=None, type='point')
Computes the values of the elementary symmetric polynomials of the n multiplier spectra of this dynamical
Can specify to instead compute the values corresponding to the elementary symmetric polynomials of the formal \( n \) multiplier spectra. The map must be defined over projective space of dimension 1. The base ring should be a number field, number field order, or a finite field or a polynomial ring or function field over a number field, number field order, or finite field.

The parameter \texttt{type} determines if the sigma are computed from the multipliers calculated at one per cycle (with multiplicity) or one per point (with multiplicity). Note that in the cycle case, a map with a cycle which collapses into multiple smaller cycles, this is still considered one cycle. In other words, if a 4-cycle collapses into a 2-cycle with multiplicity 2, there is only one multiplier used for the doubled 2-cycle when computing \( n=4 \).

**ALGORITHM:**

We use the Poisson product of the resultant of two polynomials:

\[
\text{res}(f, g) = \prod_{f(a)=0} g(a).
\]

Letting \( f \) be the polynomial defining the periodic or formal periodic points and \( g \) the polynomial \( w - f' \) for an auxiliary variable \( w \). Note that if \( f \) is a rational function, we clear denominators for \( g \).

**INPUT:**

- \( n \) – a positive integer, the period
- \texttt{formal} – (default: False) boolean; True specifies to find the values of the elementary symmetric polynomials corresponding to the formal \( n \) multiplier spectra and False specifies to instead find the values corresponding to the \( n \) multiplier spectra, which includes the multipliers of all periodic points of period \( n \)
- \texttt{embedding} – deprecated in trac ticket #23333
- \texttt{type} – (default: ’point’) string; either ’point’ or ’cycle’ depending on whether you compute with one multiplier per point or one per cycle

**OUTPUT:** a list of elements in the base ring

**EXAMPLES:**

```
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([512*x^5 - 378128*x^4*y + 76594292*x^3*y^2 - 4570550136*x^2*y^3 - 2630045017*x*y^4 + 28193217129*y^5, 512*y^5])
sage: f.sigma_invariants(1)
[19575526074450617/1048576, -90781220481450442985674132325/2147483648, -26226611149099878224381377917540931367/109951162776, -26226611079371021204196133701280271632423/549755813888, 3385232048301611650315320945076350063171417825448006778305/72057594037927936, 0]
```

```
sage: set_verbose(None)
sage: z = QQ['z'].0
sage: K = NumberField(z^4 - 4*z^2 + 1, 'z')
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: f = DynamicalSystem_projective([x^2 - 5/4*y^2, y^2])
sage: f.sigma_invariants(2, formal=False, type='cycle')
[13, 11, -25, 0]
sage: f.sigma_invariants(2, formal=False, type='point')
[12, -2, -36, 25, 0]
```
check that infinity as part of a longer cycle is handled correctly:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([y^2, x^2])
sage: f.sigma_invariants(2, type='cycle')
[12, 48, 64, 0]
sage: f.sigma_invariants(2, type='point')
[12, 48, 64, 0, 0]
sage: f.sigma_invariants(2, type='cycle', formal=True)
[0]
sage: f.sigma_invariants(2, type='point', formal=True)
[0, 0]
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([x^2, y^2, z^2])
sage: f.sigma_invariants(1)
Traceback (most recent call last):
  ...  
NotImplementedError: only implemented for dimension 1
```

```
sage: K.<w> = QuadraticField(3)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: f = DynamicalSystem_projective([x^2 - w*y^2, (1-w)*x*y])
sage: f.sigma_invariants(2, formal=False, type='cycle')
[6*w + 21, 78*w + 159, 210*w + 367, 90*w + 156]
sage: f.sigma_invariants(2, formal=False, type='point')
[6*w + 24, 96*w + 222, 444*w + 844, 720*w + 1257, 270*w + 468]
```

```
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 + x*y + y^2, y^2])
sage: f.sigma_invariants(1)
[3, 3, 1]
```

```
sage: R.<c> = QQ[]
sage: Pc.<x,y> = ProjectiveSpace(R, 1)
sage: f = DynamicalSystem_projective([x^2 + c*y^2, y^2])
sage: f.sigma_invariants(1)
[2, 4*c, 0]
sage: f.sigma_invariants(2, formal=True, type='point')
[8*c + 8, 16*c^2 + 32*c + 16]
sage: f.sigma_invariants(2, formal=True, type='cycle')
[4*c + 4]
```

doubled fixed point:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2 - 3/4*y^2, y^2])
sage: f.sigma_invariants(2, formal=True)
[2, 1]
```

doubled 2 cycle:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2 - 5/4*y^2, y^2])
sage: f.sigma_invariants(4, formal=False, type='cycle')
[170, 5195, 172700, 968615, 1439066, 638125, 0]
```
Given a set of rational points in the domain of this dynamical system, return all the rational preimages of those points.

In other words, all the rational points which have some iterate in the set points. This function repeatedly calls `rational_preimages`. If the degree is at least two, by Northcott, this is always a finite set. The map must be defined over number fields and be an endomorphism.

**INPUT:**

- `points` – a list of rational points in the domain of this map

**OUTPUT:** a list of rational points in the domain of this map

**EXAMPLES:**

```python
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([16*x^2 - 29*y^2, 16*y^2])
sage: sorted(f.all_rational_preimages([P(-1,4)]))
 (5/4 : 1), (7/4 : 1)]
```

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: f = DynamicalSystem_projective([76*x^2 - 180*x*y + 45*y^2 + 14*x*z +
   ˓→ 45*y*z - 90*z^2, 67*x^2 - 180*x*y - 157*x*z + 90*y*z, -90*z^2])
sage: sorted(f.all_rational_preimages([P(-9,-4,1)]))
[(-9 : -4 : 1), (0 : -1 : 1), (0 : 0 : 1), (0 : 1 : 1), (0 : 4 : 1),
 (1 : 0 : 1), (1 : 1 : 1), (1 : 2 : 1), (1 : 3 : 1)]
```

A non-periodic example

```python
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 + y^2, 2*x*y])
sage: sorted(f.all_rational_preimages([P(17,15)]))
[(1/3 : 1), (3/5 : 1), (5/3 : 1), (3 : 1)]
```

A number field example:

```python
sage: z = QQ['z'].0
sage: K.<w> = NumberField(z^3 + (z^2)/4 - (41/16)*z + 23/64);
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([16*x^2 - 29*y^2, 16*y^2])
sage: f.all_rational_preimages([P(16*w^2 - 29,16)])
[(-w^2 + 21/16 : 1),
 (w : 1),
 (w + 1/2 : 1),
 (w^2 + w - 33/16 : 1),
 (-w^2 - w + 25/16 : 1),
 (w^2 - 21/16 : 1),
 (-w^2 - w + 33/16 : 1),
 (-w : 1),
 (-w - 1/2 : 1),
(continues on next page)
```
(-w^2 + 29/16 : 1),
(w^2 - 29/16 : 1),
(w^2 + w - 25/16 : 1)]

sage: K.<w> = QuadraticField(3)
sage: P.<u,v> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([u^2+v^2, v^2])
sage: f.all_rational_preimages(P(4))
[(-w : 1), (w : 1)]

conjugating_set(other)

Return the set of elements in PGL that conjugates one dynamical system to the other.

Given two nonconstant rational functions of equal degree determine to see if there is an element of PGL
that conjugates one rational function to another. It does this by taking the fixed points of one map and
mapping them to all unique permutations of the fixed points of the other map. If there are not enough fixed
points the function compares the mapping between rational preimages of fixed points and the rational
preimages of the preimages of fixed points until there are enough points; such that there are n + 2 points
with all n + 1 subsets linearly independent.

ALGORITHM:

Implementing invariant set algorithm from the paper [FMV2014]. Given that the set of n th preimages of
fixed points is invariant under conjugation find all elements of PGL that take one set to another.

INPUT:

• other – a nonconstant rational function of same degree as self

OUTPUT:

Set of conjugating n + 1 by n + 1 matrices.

AUTHORS:

• Original algorithm written by Xander Faber, Michelle Manes, Bianca Viray [FMV2014].
• Implimented by Rebecca Lauren Miller, as part of GSOC 2016.

EXAMPLES:

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 - 2*y^2, y^2])
sage: m = matrix(QQbar, 2, 2, [-1, 3, 2, 1])
sage: g = f.conjugate(m)
sage: f.conjugating_set(g)
[[-1 3]
 [2 1]]

sage: K.<w> = QuadraticField(-1)
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x^2 + y^2, x*y])
sage: m = matrix(K, 2, 2, [1, 1, 2, 1])
sage: g = f.conjugate(m)
sage: f.conjugating_set(g) # long time
[[1 1] [-1 -1]
sage: K.<i> = QuadraticField(-1)
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: D8 = DynamicalSystem_projective([y^3, x^3])
sage: D8.conjugating_set(D8) # long time
[1 0] [0 1] [0 -i] [i 0] [0 -1] [-1 0] [-i 0] [0 i]
[0 1], [1 0], [1 0], [0 1], [0 1], [0 1], [0 1], [1 0]

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: D8 = DynamicalSystem_projective([y^2, x^2])
sage: D8.conjugating_set(D8)
Traceback (most recent call last):
... ValueError: not enough rational preimages

sage: P.<x,y> = ProjectiveSpace(GF(7),1)
sage: D6 = DynamicalSystem_projective([y^2, x^2])
sage: D6.conjugating_set(D6)
[1 0] [0 1] [0 2] [4 0] [2 0] [0 4]
[0 1], [1 0], [1 0], [0 1], [0 1], [1 0]

sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: f = DynamicalSystem_projective([x^2 + x*z, y^2, z^2])
sage: f.conjugating_set(f) # long time
[1 0 0]
[0 1 0]
[0 0 1]

connected_rational_component \(P, n=0\)

Computes the connected component of a rational preperiodic point \(P\) by this dynamical system.

Will work for non-preperiodic points if \(n\) is positive. Otherwise this will not terminate.

INPUT:

- \(P\) – a rational preperiodic point of this map
- \(n\) – (default: 0) integer; maximum distance from \(P\) to branch out; a value of 0 indicates no bound

OUTPUT:

A list of points connected to \(P\) up to the specified distance.

EXAMPLES:
sage: P = PS([w,1])
sage: f.connected_rational_component(P)
[(w : 1),
 (w^2 - 29/16 : 1),
 (-w^2 - w + 25/16 : 1),
 (w^2 + w - 25/16 : 1),
 (-w : 1),
 (-w^2 + 29/16 : 1),
 (w + 1/2 : 1),
 (-w - 1/2 : 1),
 (-w^2 + 21/16 : 1),
 (w^2 - 21/16 : 1),
 (w^2 + w - 33/16 : 1),
 (-w^2 - w + 33/16 : 1)]

sage: PS.<x,y,z> = ProjectiveSpace(2,QQ)
sage: f = DynamicalSystem_projective([x^2 - 21/16*z^2, y^2-2*z^2, z^2])
sage: P = PS([17/16,7/4,1])
sage: f.connected_rational_component(P,3)
[(17/16 : 7/4 : 1),
 (-47/256 : 17/16 : 1),
 (-83807/65536 : -223/256 : 1),
 (-17/16 : -7/4 : 1),
 (17/16 : -7/4 : 1),
 (1386468673/4294967296 : -81343/65536 : 1),
 (-47/256 : -17/16 : 1),
 (47/256 : -17/16 : 1),
 (47/256 : 17/16 : 1),
 (-1/2 : -1/2 : 1),
 (-1/2 : 1/2 : 1),
 (1/2 : -1/2 : 1),
 (1/2 : 1/2 : 1)]

is_conjugate(other)
Return whether or not two dynamical systems are conjugate.

ALGORITHM:
Implementing invariant set algorithm from the paper [FMV2014]. Given that the set of \( n \) th preimages is invariant under conjugation this function finds whether two maps are conjugate.

INPUT:

• other – a nonconstant rational function of same degree as self

OUTPUT: boolean

AUTHORS:

• Original algorithm written by Xander Faber, Michelle Manes, Bianca Viray [FMV2014].
• Implimented by Rebecca Lauren Miller as part of GSOC 2016.

EXAMPLES:

sage: K.<w> = CyclotomicField(3)
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: D8 = DynamicalSystem_projective([y^2, x^2])
```python
sage: D8.is_conjugate(D8)
True

sage: setVerbose(None)

sage: P.<x,y> = ProjectiveSpace(QQbar, 1)
sage: f = DynamicalSystem_projective([x^2 + x*y, y^2])
sage: m = matrix(QQbar, 2, 2, [1, 1, 2, 1])
sage: g = f.conjugate(m)
sage: f.is_conjugate(g) # long time
True

sage: P.<x,y> = ProjectiveSpace(GF(5), 1)
sage: f = DynamicalSystem_projective([x^3 + x*y^2, y^3])
sage: m = matrix(GF(5), 2, 2, [1, 3, 2, 9])
sage: g = f.conjugate(m)
sage: f.is_conjugate(g)
True

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2 + x*y, y^2])
sage: g = DynamicalSystem_projective([x^3 + x^2*y, y^3])
sage: f.is_conjugate(g)
False

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2 + x*y, y^2])
sage: g = DynamicalSystem_projective([x^2 - 2*y^2, y^2])
sage: f.is_conjugate(g)
False

is_polynomial()
Check to see if the dynamical system has a totally ramified fixed point.
The function must be defined over an absolute number field or a finite field.

OUTPUT: boolean

EXAMPLES:

sage: R.<x> = QQ[]
sage: K.<w> = QuadraticField(7)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: f = DynamicalSystem_projective([x**2 + 2*x*y - 5*y**2, 2*x*y])
sage: f.is_polynomial()
False

sage: R.<x> = QQ[]
sage: K.<w> = QuadraticField(7)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: f = DynamicalSystem_projective([x**2 - 7*x*y, 2*y**2])
sage: m = matrix(K, 2, 2, [w, 1, 0, 1])
sage: f = f.conjugate(m)
sage: f.is_polynomial()
True
```

6.3. Dynamical systems on projective schemes 229
sage: K.<w> = QuadraticField(4/27)
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x**3 + w*y**3, x*y**2])
sage: f.is_polynomial()
False

sage: K = GF(3**2, prefix='w')
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x**2 + K.gen()*y**2, x*y])
sage: f.is_polynomial()
False

sage: PS.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([6*x^2+12*x*y+7*y^2, 12*x*y + 42*y^2])
sage: f.is_polynomial()
False

**lift_to_rational_periodic** *(points_modp, B=None)*

Given a list of points in projective space over \( \mathbb{F}_p \), determine if they lift to \( \mathbb{Q} \)-rational periodic points.

The map must be an endomorphism of projective space defined over \( \mathbb{Q} \).

**ALGORITHM:**

Use Hensel lifting to find a \( p \)-adic approximation for that rational point. The accuracy needed is determined by the height bound \( B \). Then apply the LLL algorithm to determine if the lift corresponds to a rational point.

If the point is a point of high multiplicity (multiplier 1), the procedure can be very slow.

**INPUT:**

- **points_modp** – a list or tuple of pairs containing a point in projective space over \( \mathbb{F}_p \) and the possible period
- **B** – (optional) a positive integer; the height bound for a rational preperiodic point

**OUTPUT:** a list of projective points

**EXAMPLES:**

```sage
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 - y^2, y^2])
sage: f.lift_to_rational_periodic([[P(0,1).change_ring(GF(7)), 4]])
[[[0 : 1], 2]]
```

There may be multiple points in the lift.

```sage
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([-5*x^2 + 4*y^2, 4*x*y])
sage: f.lift_to_rational_periodic([[P(1,0).change_ring(GF(3)), 1]]) # long time
[[[1 : 0], [2/3 : 1], [-2/3 : 1]]]
```

```sage
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([16*x^2 - 29*y^2, 16*y^2])
sage: f.lift_to_rational_periodic([[P(3,1).change_ring(GF(13)), 3]])
[[[-1/4 : 1], 3]]
```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([76*x^2 - 180*x*y + 45*y^2 + 14*x*z + 45*y*z - 90*z^2, 67*x^2 - 180*x*y - 157*x*z + 90*y*z, -90*z^2])
sage: f.lift_to_rational_periodic([[P(14,19,1).change_ring(GF(23)), 9]])
# long time
[[(-9 : -4 : 1), 9]]

normal_form (return_conjugation=False)

Return a normal form in the moduli space of dynamical systems.

Currently implemented only for polynomials. The totally ramified fixed point is moved to infinity and the map is conjugated to the form \(x^n + a_{n-2}x^{n-2} + \cdots + a_0\). Note that for finite fields we can only remove the \((n - 1)\)-st term when the characteristic does not divide \(n\).

INPUT:

- return_conjugation – (default: False) boolean; if True, then return the conjugation element of PGL along with the embedding into the new field

OUTPUT:

- SchemeMorphism_polynomial
  - (optional) an element of PGL as a matrix
  - (optional) the field embedding

EXAMPLES:

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2 + 2*x*y - 5*x^2, 2*x*y])
sage: f.normal_form()
Traceback (most recent call last):
... Not ImplementedError: map is not a polynomial

sage: R.<x> = QQ[]
sage: K.<w> = NumberField(x^2 - 5)
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x^2 + w*x*y, y^2])
sage: g,m,psi = f.normal_form(return_conjugation = True);m
[ 1 -1/2 *w]
[ 0 1]
sage: f.change_ring(psi).conjugate(m) == g
True

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([13*x^2 + 4*x*y + 3*y^2, 5*y^2])
sage: f.normal_form()
Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to
          (5*x^2 + 9*y^2 : 5*y^2)

sage: K = GF(3^3, prefix = 'w')
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x^3 + 2*x^2*y + 2*x*y^2 + K.gen()*y^3, y^3])
sage: f.normal_form()
Dynamical System of Projective Space of dimension 1 over Finite Field in w of size 3^3
(continues on next page)
Defn: Defined on coordinates by sending \((x : y)\) to
\((x^3 + x^2y + x*y^2 + (-w3)*y^3 : y^3)\)

**rational_periodic_points(**\*kwds\*)**

Determine the set of rational periodic points for this dynamical system.

The map must be defined over \(\mathbb{Q}\) and be an endomorphism of projective space. If the map is a polynomial endomorphism of \(\mathbb{P}^1\), i.e. has a totally ramified fixed point, then the base ring can be an absolute number field. This is done by passing to the Weil restriction.

The default parameter values are typically good choices for \(\mathbb{P}^1\). If you are having trouble getting a particular map to finish, try first computing the possible periods, then try various different **lifting_prime** values.

**ALGORITHM:**

Modulo each prime of good reduction \(p\) determine the set of periodic points modulo \(p\). For each cycle modulo \(p\) compute the set of possible periods (\(mp^r\)). Take the intersection of the list of possible periods modulo several primes of good reduction to get a possible list of minimal periods of rational periodic points. Take each point modulo \(p\) associated to each of these possible periods and try to lift it to a rational point with a combination of \(p\)-adic approximation and the LLL basis reduction algorithm.

See [Hutz2015].

**INPUT:**

**kwds:**

- **prime_bound** – (default: \([1,20]\)) a pair (list or tuple) of positive integers that represent the limits of primes to use in the reduction step or an integer that represents the upper bound
- **lifting_prime** – (default: 23) a prime integer; argument that specifies modulo which prime to try and perform the lifting
- **periods** – (optional) a list of positive integers that is the list of possible periods
- **bad_primes** – (optional) a list or tuple of integer primes; the primes of bad reduction
- **ncpus** – (default: all cpus) number of cpus to use in parallel

**OUTPUT:** a list of rational points in projective space

**EXAMPLES:**

```
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2-3/4*y^2, y^2])
sage: sorted(f.rational_periodic_points(prime_bound=20, lifting_prime=7))
# long time
[(-1/2 : 1), (1 : 0), (3/2 : 1)]
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: f = DynamicalSystem_projective([2*x^3 - 50*x*z^2 + 24*z^3,
....: 5*y^3 - 53*y*z^2 + 24*z^3, 24*z^3])
sage: sorted(f.rational_periodic_points(prime_bound=[1,20]))
# long time
[(-3 : -1 : 1), (-3 : 0 : 1), (-3 : 1 : 1), (-3 : 3 : 1), (-1 : -1 : 1),
(-1 : 0 : 1), (-1 : 1 : 1), (-1 : 3 : 1), (0 : 1 : 0), (1 : -1 : 1),
(1 : 0 : 0), (1 : 0 : 1), (1 : 1 : 1), (1 : 3 : 1), (3 : -1 : 1),
(3 : 0 : 1), (3 : 1 : 1), (3 : 3 : 1), (5 : -1 : 1), (5 : 0 : 1),
(5 : 1 : 1), (5 : 3 : 1)]
```
```python
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([-5*x^2 + 4*y^2, 4*x*y])
sage: sorted(f.rational_periodic_points()) # long time
[(-2 : 1), (-2/3 : 1), (2/3 : 1), (1 : 0), (2 : 1)]
```

```python
sage: R.<x> = QQ[]
sage: K.<w> = NumberField(x^2-x+1)
sage: P.<u,v> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([u^2 + v^2,v^2])
sage: f.rational_periodic_points()
[(w : 1), (1 : 0), (-w + 1 : 1)]
```

```python
sage: R.<x> = QQ[]
sage: K.<w> = NumberField(x^2-x+1)
sage: P.<u,v> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([u^2+v^2,u*v])
sage: f.rational_periodic_points()
Traceback (most recent call last):
... 
NotImplementedError: rational periodic points for number fields only
←implemented for polynomials
```

**rational_preperiodic_graph(**kwsd)**

Determine the directed graph of the rational preperiodic points for this dynamical system.

The map must be defined over $\mathbb{Q}$ and be an endomorphism of projective space. If this map is a polynomial endomorphism of $\mathbb{P}^1$, i.e. has a totally ramified fixed point, then the base ring can be an absolute number field. This is done by passing to the Weil restriction.

**ALGORITHM:**

- Determines the list of possible periods.
- Determines the rational periodic points from the possible periods.
- Determines the rational preperiodic points from the rational periodic points by determining rational preimages.

**INPUT:**

kwsd:

- **prime_bound** – (default: $[1, 20]$) a pair (list or tuple) of positive integers that represent the limits of primes to use in the reduction step or an integer that represents the upper bound
- **lifting_prime** – (default: 23) a prime integer; specifies modulo which prime to try and perform the lifting
- **periods** – (optional) a list of positive integers that is the list of possible periods
- **bad_primes** – (optional) a list or tuple of integer primes; the primes of bad reduction
- **ncpus** – (default: all cpus) number of cpus to use in parallel

**OUTPUT:**

A digraph representing the orbits of the rational preperiodic points in projective space.

**EXAMPLES:**
sage: PS.<x,y> = ProjectiveSpace(1,QQ)
sage: f = DynamicalSystem_projective([7*x^2 - 28*y^2, 24*x*y])
sage: f.rational_preperiodic_graph()
Looped digraph on 12 vertices

sage: PS.<x,y> = ProjectiveSpace(1,QQ)
sage: f = DynamicalSystem_projective([-3/2*x^3 +19/6*x*y^2, y^3])
sage: f.rational_preperiodic_graph(prime_bound=[1,8])
Looped digraph on 12 vertices

sage: PS.<x,y,z> = ProjectiveSpace(2,QQ)
sage: f = DynamicalSystem_projective([2*x^3 - 50*x*z^2 + 24*z^3,
.....: 5*y^3 - 53*y*z^2 + 24*z^3, 24*z^3])
sage: f.rational_preperiodic_graph(prime_bound=[1,11], lifting_prime=13) # long time
Looped digraph on 30 vertices

rational_preperiodic_points(**kwds)

Determine the set of rational preperiodic points for this dynamical system.

The map must be defined over \( \mathbb{Q} \) and be an endomorphism of projective space. If the map is a polynomial endomorphism of \( \mathbb{P}^1 \), i.e. has a totally ramified fixed point, then the base ring can be an absolute number field. This is done by passing to the Weil restriction.

The default parameter values are typically good choices for \( \mathbb{P}^1 \). If you are having trouble getting a particular map to finish, try first computing the possible periods, then try various different values for \( \text{lifting_prime} \).

ALGORITHM:

• Determines the list of possible periods.
• Determines the rational periodic points from the possible periods.
• Determines the rational preperiodic points from the rational periodic points by determining rational preimages.

INPUT:

kwds:

• prime_bound – (default: [1, 20]) a pair (list or tuple) of positive integers that represent the limits of primes to use in the reduction step or an integer that represents the upper bound
• lifting_prime – (default: 23) a prime integer; specifies modulo which prime to try and perform the lifting
• periods – (optional) a list of positive integers that is the list of possible periods
• bad_primes – (optional) a list or tuple of integer primes; the primes of bad reduction
• ncpus – (default: all cpus) number of cpus to use in parallel

OUTPUT: a list of rational points in projective space

EXAMPLES:
sage: PS.<x,y> = ProjectiveSpace(1,QQ)
sage: f = DynamicalSystem_projective([x^2 -y^2, 3*x+y])
sage: sorted(f.rational_preperiodic_points())
[(-2 : 1), (-1 : 1), (-1/2 : 1), (0 : 1), (1/2 : 1), (1 : 0), (1 : 1), (2 : 1)]

sage: PS.<x,y> = ProjectiveSpace(1,QQ)
sage: f = DynamicalSystem_projective([5*x^3 - 53*x*y^2 + 24*y^3, 24*y^3])
sage: sorted(f.rational_preperiodic_points(prime_bound=10))
[(-1 : 1), (0 : 1), (1 : 0), (1 : 1), (3 : 1)]

sage: PS.<x,y,z> = ProjectiveSpace(2,QQ)
sage: f = DynamicalSystem_projective([x^2 - 21/16*z^2, y^2-2*z^2, z^2])
sage: sorted(f.rational_preperiodic_points(prime_bound=[1,8], lifting_prime=7, periods=[2]))
# long time

sage: K.<w> = QuadraticField(33)
sage: PS.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x^2-71/48*y^2, y^2])
sage: sorted(f.rational_preperiodic_points())
# long time
[(-1/12*w - 1 : 1), (-1/6*w - 1/4 : 1), (-1/12*w - 1/2 : 1), (-1/6*w + 1/4 : 1), (1/12*w - 1 : 1), (1/12*w - 1/2 : 1), (-1/12*w + 1/2 : 1), (-1/12*w + 1 : 1), (1/6*w - 1/4 : 1), (1/12*w + 1/2 : 1), (1 : 0), (1/6*w + 1/4 : 1), (1/12*w + 1 : 1)]

class sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_finite_field

Bases: sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field, sage.schemes.projective.projective_morphism.SchemeMorphism_polynomial_projective_space_finite_field

automorphism_group (absolute=False, iso_type=False, return_functions=False)

Return the subgroup of PGL2 that is the automorphism group of this dynamical system.

Only for dimension 1. The automorphism group is the set of PGL2 elements that fixed the map under conjugation. See [FMV2014] for the algorithm.

INPUT:

- absolute-- (default: False) boolean; if True, then return the absolute automorphism group and a field of definition.
• iso_type – (default: False) boolean; if True, then return the isomorphism type of the automorphism group
• return_functions – (default: False) boolean; True returns elements as linear fractional transformations and False returns elements as PGL2 matrices

OUTPUT: a list of elements of the automorphism group

AUTHORS:
• Original algorithm written by Xander Faber, Michelle Manes, Bianca Viray
• Modified by Joao Alberto de Faria, Ben Hutz, Bianca Thompson

EXAMPLES:

```
sage: R.<x,y> = ProjectiveSpace(GF(7^3,'t'),1)
sage: f = DynamicalSystem_projective([x^2-y^2, x*y])
sage: f.automorphism_group()
[ [1 0] [6 0]
[0 1], [0 1] ]
```

```
sage: R.<x,y> = ProjectiveSpace(GF(3^2,'t'),1)
sage: f = DynamicalSystem_projective([x^3,y^3])
sage: f.automorphism_group(return_functions=True, iso_type=True) # long time
([x,
x/(x + 1),
2*x/(x + 2),
2/(x + 2),
(x + 2)/x,
(2*x + 2)/x,
2/(x + 1),
x + 1,
x + 2,
x/(x + 2),
2*x/(x + 1),
2*x,
1/x,
2*x + 1,
2*x + 2,
(x + 2)/(x + 1),
2/x,
(2*x + 2)/(x + 2),
(x + 1)/(x + 2),
(2*x + 1)/x,
1/(x + 1),
1/(x + 2),
(2*x + 1)/(x + 1),
(x + 1)/x],
'PGL(2,3)')
```

```
sage: R.<x,y> = ProjectiveSpace(GF(2^5,'t'),1)
sage: f = DynamicalSystem_projective([x^5,y^5])
sage: f.automorphism_group(return_functions=True, iso_type=True)
(['x, 1/x], 'Cyclic of order 2')
```

```
cyclegraph()

Return the digraph of all orbits of this dynamical system.

Over a finite field this is a finite graph. For subscheme domains, only points on the subscheme whose image are also on the subscheme are in the digraph.

OUTPUT: a digraph

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(GF(13),1)
sage: f = DynamicalSystem_projective([x^2-y^2, y^2])
sage: f.cyclegraph()
Looped digraph on 14 vertices
```

```
sage: P.<x,y,z> = ProjectiveSpace(GF(3^2,'t'),2)
sage: f = DynamicalSystem_projective([x^2+y^2, y^2, z^2+y*z])
sage: f.cyclegraph()
Looped digraph on 91 vertices
```

```
sage: P.<x,y,z> = ProjectiveSpace(GF(7),2)
sage: X = P.subscheme(x^2-y^2)
sage: f = DynamicalSystem_projective([x^2, y^2, z^2], domain=X)
sage: f.cyclegraph()
Looped digraph on 15 vertices
```

```
sage: P.<x,y,z> = ProjectiveSpace(GF(3),2)
sage: f = DynamicalSystem_projective([x*z-y^2, x^2-y^2, y^2-z^2])
sage: f.cyclegraph()
Looped digraph on 13 vertices
```

```
sage: P.<x,y,z> = ProjectiveSpace(GF(3),2)
sage: X = P.subscheme([x-y])
sage: f = DynamicalSystem_projective([x^2-y^2, x^2-y^2, y^2-z^2], domain=X)
sage: f.cyclegraph()
Looped digraph on 4 vertices
```

orbit_structure($P$)

Return the pair $[m, n]$, where $m$ is the preperiod and $n$ is the period of the point $P$ by this dynamical system.

Every point is preperiodic over a finite field so every point will be preperiodic.

INPUT:

- $P$ – a point in the domain of this map

OUTPUT: a list $[m, n]$ of integers

EXAMPLES:
sage: P.<x,y,z> = ProjectiveSpace(GF(5),2)
sage: f = DynamicalSystem_projective([x^2 + y^2, y^2, z^2 + y*z], domain=P)
sage: f.orbit_structure(P(2,1,2))
[0, 6]

sage: P.<x,y,z> = ProjectiveSpace(GF(7),2)
sage: X = P.subscheme(x^2-y^2)
sage: f = DynamicalSystem_projective([x^2, y^2, z^2], domain=X)
sage: f.orbit_structure(X(1,1,2))
[0, 2]

sage: P.<x,y> = ProjectiveSpace(GF(13),1)
sage: f = DynamicalSystem_projective([x^2 - y^2, y^2], domain=P)
sage: f.orbit_structure(P(3,4))
[2, 3]

sage: R.<t> = GF(13^3)
sage: P.<x,y> = ProjectiveSpace(R,1)
sage: f = DynamicalSystem_projective([x^2 - y^2, y^2], domain=P)
sage: f.orbit_structure(P(t, 4))
[11, 6]

possible_periods (return_points=False)

Return the list of possible minimal periods of a periodic point over \( Q \) and (optionally) a point in each cycle.

ALGORITHM:
See [Hutz2009].

INPUT:

- return_points – (default: False) boolean; if True, then return the points as well as the possible periods

OUTPUT:

A list of positive integers, or a list of pairs of projective points and periods if return_points is True.

EXAMPLES:

sage: P.<x,y> = ProjectiveSpace(GF(23),1)
sage: f = DynamicalSystem_projective([x^2-y^2, y^2])
sage: f.possible_periods()
[1, 5, 11, 22, 110]

sage: P.<x,y> = ProjectiveSpace(GF(13),1)
sage: f = DynamicalSystem_projective([x^2-y^2, y^2])
sage: sorted(f.possible_periods(True))
[[0 : 1), 2], [(1 : 0), 1], [(3 : 1), 3], [(3 : 1), 36]]

sage: PS.<x,y,z> = ProjectiveSpace(2,GF(7))
sage: f = DynamicalSystem_projective([-360*x^3 + 760*x*z^2, y^3 - 604*y*z^2 + 240*z^3, 240*z^3])
sage: f.possible_periods()
[1, 2, 4, 6, 12, 14, 28, 42, 84]
Todo:
- do not return duplicate points
- improve hash to reduce memory of point-table

6.4 Dynamical systems for products of projective spaces

This class builds on the product projective space class. The main constructor functions are given by
\texttt{DynamicalSystem} and \texttt{DynamicalSystem\_projective}. The constructors function can take either polynomials or a morphism from which to construct a dynamical system.

The must be specified.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: P1xP1.<x,y,u,v> = ProductProjectiveSpaces(QQ, [1, 1])
sage: DynamicalSystem\_projective([x^2*u, y^2*v, x*v^2, y*u^2], domain=P1xP1)
Dynamical System of Product of projective spaces P^1 x P^1 over Rational Field
  Defn: Defined by sending (x : y , u : v) to
         (x^2*u : y^2*v , x*v^2 : y*u^2).
\end{verbatim}

\textbf{class} sage.dynamics.arithmetic_dynamics.product_projective_ds.DynamicalSystem\_product_projective

\textbf{INPUT:}
- \texttt{polys} – a list of \( n_1 + \cdots + n_r \) multi-homogeneous polynomials, all of which should have the same parent
- \texttt{domain} – a projective scheme embedded in \( P^{n_1-1} \times \cdots \times P^{n_r-1} \)

\textbf{EXAMPLES:}

\begin{verbatim}
sage: T.<x,y,z,w,u> = ProductProjectiveSpaces([2, 1], QQ)
sage: DynamicalSystem\_projective([x^2, y^2, z^2, w^2, u^2], domain=T)
Dynamical System of Product of projective spaces P^2 x P^1 over Rational Field
  Defn: Defined by sending (x : y , z , w : u) to
         (x^2 : y^2 : z^2 , w^2 : u^2).
\end{verbatim}

\texttt{nth\_iterate}(P, n, normalize=False)

Return the \( n \)-th iterate of \( P \) by this dynamical system.

If \texttt{normalize} is \texttt{True}, then the coordinates are automatically normalized.
Todo: Is there a more efficient way to do this?

INPUT:

- \( P \) – a point in \( \text{self.domain}() \)
- \( n \) – a positive integer
- \( \text{normalize} \) – (default: False) boolean

OUTPUT: A point in \( \text{self.ccodomain}() \)

EXAMPLES:

```
sage: Z.<a,b,x,y,z> = ProductProjectiveSpaces([1, 2], QQ)
sage: f = DynamicalSystem_projective([a^3, b^3 + a*b^2, x^2, y^2 - z^2, z*y], domain=Z)
sage: P = Z([1, 1, 1, 1, 1])
sage: f.nth_iterate(P, 3)
(1/1872 : 1 , 1 : 1 : 0)
```

```
sage: Z.<a,b,x,y> = ProductProjectiveSpaces([1, 1], ZZ)
sage: f = DynamicalSystem_projective([a*b, b^2, x^3 - y^3, y^2*x], domain=Z)
sage: P = Z([2, 6, 2, 4])
sage: f.nth_iterate(P, 2, normalize = True)
(1 : 3 , 407 : 112)
```

**nth_iterate_map** \((n)\)

Return the \( n \)th iterate of this dynamical system.

ALGORITHM:

Uses a form of successive squaring to reduce computations.

Todo: This could be improved.

INPUT:

- \( n \) – a positive integer

OUTPUT: A dynamical system of products of projective spaces

EXAMPLES:

```
sage: Z.<a,b,x,y,z> = ProductProjectiveSpaces([1, 2], QQ)
sage: f = DynamicalSystem_projective([a^3, b^3 + a*b^2, x^2, y^2 - z^2, z*y], domain=Z)
sage: f.nth_iterate_map(3)
Dynamical System of Product of projective spaces P^1 x P^2 over Rational Field
  Defn: Defined by sending (a : b , x : y : z) to (a^27 : b^27 , x^8 : y^8 : z^8).
```

**orbit** \((P, N, **kwds)\)

Return the orbit of \( P \) by this dynamical system.

Let \( F \) be this dynamical system. If \( N \) is an integer return \([P, F(P), \ldots, F^N(P)]\).

If \( N \) is a list or tuple \( N = [m, k] \) return \([F^m(P), \ldots, F^k(P)]\). Automatically normalize the points if normalize == True. Perform the checks on point initialize if check==True.
INPUT:
- \(P\) – a point in \(\text{self.domain}()\)
- \(N\) – a non-negative integer or list or tuple of two non-negative integers

kwds:
- \(\text{check} - \) (default: True) boolean
- \(\text{normalize} - \) (default: False) boolean

OUTPUT: a list of points in \(\text{self.codomain}()\)

EXAMPLES:

```python
sage: Z.<a,b,x,y,z> = ProductProjectiveSpaces([1, 2], QQ)
sage: f = DynamicalSystem_projective([a^3, b^3 + a*b^2, x^2, y^2 - z^2, z*y], domain=Z)
sage: P = Z([1, 1, 1, 1, 1])
sage: f.orbit(P, 3)
[(1 : 1 : 1 : 1), (1/2 : 1 : 1 : 0 : 1), (1/12 : 1 : -1 : 1 : 0), (1/1872 : ... : 5105408)]
```

```python
sage: Z.<a,b,x,y> = ProductProjectiveSpaces([1, 1], ZZ)
sage: f = DynamicalSystem_projective([a*b, b^2, x^3 - y^3, y^2*x], domain=Z)
sage: P = Z([2, 6, 2, 4])
sage: f.orbit(P, 3, normalize=True)
```

6.5 Wehler K3 Surfaces

AUTHORS:
- Ben Hutz (11-2012)
- Joao Alberto de Faria (10-2013)

Todo: Hasse-Weil Zeta Function
Picard Number
Number Fields

REFERENCES: [FH2015], [CS1996], [Weh1998], [Hutz2007]

sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface \((\text{polys})\)

Defines a K3 Surface over \(\mathbb{P}^2 \times \mathbb{P}^2\) defined as the intersection of a bilinear and biquadratic form. [Weh1998]

INPUT: Bilinear and biquadratic polynomials as a tuple or list

OUTPUT: \(\text{WehlerK3Surface}\)

EXAMPLES:

```python
sage: PP.<x0,x1, x2, y0, y1, y2> = ProductProjectiveSpaces([2, 2],QQ)
sage: L = x0*y0 + x1*y1 - x2*y2
sage: Q = x0*x1*y1^2 + x2^2*y0*y2
```
Wehler K3 Surface (\([L, Q]\))

Closed subscheme of Product of projective spaces \(P^2 \times P^2\) over Rational Field defined by:

\[
\begin{align*}
&x_0 y_0 + x_1 y_1 - x_2 y_2, \\
&x_0 x_1 y_1^2 + x_2^2 y_0 y_2
\end{align*}
\]

```
sage: WehlerK3Surface([L, Q])
Closed subscheme of Product of projective spaces P^2 x P^2 over Rational Field defined by:
  x0*y0 + x1*y1 - x2*y2,
  x0*x1*y1^2 + x2^2*y0*y2
```

class sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_field(polys)
Bases: sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring

class sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_finite_field(polys)
Bases: sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_field

cardinality()
Counts the total number of points on the K3 surface.

**ALGORITHM:**
Enumerate points over \(\mathbb{F}^2\), and then count the points on the fiber of each of those points.

**OUTPUT:** Integer - total number of points on the surface

**EXAMPLES:**
```
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], GF(7))
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 
                 3*x0*x1*y0*y1 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 
                 - 4*x1*x2*y1^2 + 5*x0*x2*y0*y2 -4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 
                 + x0*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: X.cardinality()
sage: X.cardinality()
55
```

class sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring(polys)
Bases: sage.schemes.product_projective.subscheme.AlgebraicScheme_subscheme_product_projective

A K3 surface in \(\mathbb{P}^2 \times \mathbb{P}^2\) defined as the intersection of a bilinear and biquadratic form. [Weh1998]

**EXAMPLES:**
```
sage: R.<x,y,z,u,v,w> = PolynomialRing(QQ, 6)
sage: L = x*u - y*v
sage: Q = x*y*v^2 + z^2*u*w
sage: WehlerK3Surface([L, Q])
Closed subscheme of Product of projective spaces P^2 x P^2 over Rational Field defined by:
  x*u - y*v,
  x*y*v^2 + z^2*u*w
```

gpoly (component, k)

Returns the G polynomials \(G^*_k\).

They are defined as: \(G^*_k = (L_j^*)^2 Q^*_i - L_i^* L_j^* Q^*_i + (L_j^*)^2 Q^*_j\), where \(i, j, k\) is some permutation of \((0, 1, 2)\) and \(*\) is either \(x\) (Component = 1) or \(y\) (Component = 0).

**INPUT:**
- component - Integer: 0 or 1
- k - Integer: 0, 1 or 2
OUTPUT: polynomial in terms of either y (Component = 0) or x (Component = 1)

EXAMPLES:

```sage
sage: R.<x0,x1,x2,y0,y1,y2> = PolynomialRing(ZZ, 6)
sage: Y = x0*y0 + x1*y1 - x2*y2
sage: Z = x0^2*y0*y1 + x0^2*y2^2 - x0*x1*y1*y2 + x1^2*y2*y1 \n    + x2^2*y2^2 + x2^2*y1^2 + x1^2*y2^2
sage: X = WehlerK3Surface([Z, Y])
sage: X.Gpoly(1, 0)
x0^2*x1^2 + x1^4 - x0*x1^2*x2 + x1^3*x2 + x1^2*x2^2 + x2^4
```

**Hpoly** \( (\text{component}, i, j) \)

Returns the H polynomials defined as \( H_{ij}^* \).

This polynomial is defined by:

\[
H_{ij}^* = 2 L_i^* L_j^* Q_{jk}^* - L_i^* L_k^* Q_{jk}^* - L_j^* L_k^* Q_{ik}^* + (L_k^*)^2 Q_{ij}^*
\]

where \( i, j, k \) is some permutation of \( (0, 1, 2) \) and * is either y (Component = 0) or x (Component = 1).

INPUT:
- component - Integer: 0 or 1
- i - Integer: 0, 1 or 2
- j - Integer: 0, 1 or 2

OUTPUT: polynomial in terms of either y (Component = 0) or x (Component = 1)

EXAMPLES:

```sage
sage: R.<x0,x1,x2,y0,y1,y2> = PolynomialRing(ZZ, 6)
sage: Y = x0*y0 + x1*y1 - x2*y2
sage: Z = x0^2*y0*y1 + x0^2*y2^2 - x0*x1*y1*y2 + x1^2*y2*y1 \n    + x2^2*y2^2 + x2^2*y1^2 + x1^2*y2^2
sage: X = WehlerK3Surface([Z, Y])
sage: X.Hpoly(0, 1, 0)
2*y0*y1^3 + 2*y0*y1*y2^2 - y1*y2^3
```

**Lxa** \( (a) \)

Function will return the L polynomial defining the fiber, given by \( L_a^* \).

This polynomial is defined as:

\[
L_a^* = \{(a, y) \in \mathbb{P}^2 \times \mathbb{P}^2 : L(a, y) = 0\}.
\]

Notation and definition from: [CS1996]

INPUT: a - Point in \( \mathbb{P}^2 \)

OUTPUT: A polynomial representing the fiber

EXAMPLES:

```sage
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 \n    + 3*x0*x1*y0*y1 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - 4*x1*x2*y1^2 \n    + 5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 \n    + 4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
```

(continues on next page)
Lyb\((b)\)
Function will return a fiber by \(L^y_b\).

This polynomial is defined as:
\[L^y_b = \{(x, b) \in \mathbb{P}^2 \times \mathbb{P}^2 : L(x, b) = 0\}.\]

Notation and definition from: [CS1996]

INPUT: \(b\) - Point in projective space

OUTPUT: A polynomial representing the fiber

EXAMPLES:

```
sage: T = PP(1, 1, 0, 1, 0, 0)
sage: X.Lxa(T[0])
y0 + y1
```

Qxa\((a)\)
Function will return the Q polynomial defining a fiber given by \(Q^x_a\).

This polynomial is defined as:
\[Q^x_a = \{(a, y) \in \mathbb{P}^2 \times \mathbb{P}^2 : Q(a, y) = 0\}.\]

Notation and definition from: [CS1996]

INPUT: \(a\) - Point in \(\mathbb{P}^2\)

OUTPUT: A polynomial representing the fiber

EXAMPLES:

```
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 3*x0*x1*y0*y1
   + 5*x0*x2*y0*y2
   - 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: T = PP(1, 1, 0, 1, 0, 0)
sage: X.Qxa(T[0])
x0
```

Qyb\((b)\)
Function will return a fiber by \(Q^y_b\).
This polynomial is defined as:

\[ Q_b^x = \{(x, b) \in \mathbb{P}^2 \times \mathbb{P}^2 : Q(x, b) = 0\}. \]

Notation and definition from: [CS1996]

INPUT: b - Point in projective space

OUTPUT: A polynomial representing the fiber

EXAMPLES:

```
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 3*x0*x1*y0*y1 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 - 4*x1*x2*y1^2 + 7*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: T = PP(1, 1, 0, 1, 0, 0)
sage: X.Qyb(T[1])
x0^2 + 3*x0*x1 + x1^2
```

Ramification_poly (i)

Function will return the Ramification polynomial \( g^* \).

This polynomial is defined by:

\[ g^* = \frac{(H_1^*)^2 - 4G_1^*G_1^*}{(L_1^*)^2}. \]

The roots of this polynomial will either be degenerate fibers or fixed points of the involutions \( \sigma_x \) or \( \sigma_y \) for more information, see [CS1996].

INPUT: i - Integer, either 0 (polynomial in y) or 1 (polynomial in x)

OUTPUT: Polynomial in the coordinate ring of the ambient space

EXAMPLES:

```
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = 8*y0^5*y1 - 24*y0^4*y1^2 + 48*y0^2*y1^4 - 16*y0*y1^5 + y1^6 + 84*y0^3*y1^2*y2 + 46*y0^2*y1^3*y2 - 20*y0*y1^4*y2 + 16*y1^5*y2 + 53*y0^4*y2^2 + 56*y0^3*y1*y2^2 - 32*y0^2*y2^3 - 80*y0*y1^3*y2^2 - 92*y1^4*y2^2 - 12*y0^2*y1*y2^3 - 168*y0*y1^2*y2^3 - 122*y1^3*y2^3 + 14*y0^2*y2^4 + 8*y0*y1*y2^4 - 112*y1^2 - 2*y2^4 + y2^6
```

Sxa (a)

Function will return fiber by \( S_x^a \).

This function is defined as:

\[ S_x^a = L_x^a \cap Q_x^a. \]

Notation and definition from: [CS1996]
INPUT: a - Point in $\mathbb{P}^2$

OUTPUT: A subscheme representing the fiber

EXAMPLES:

```
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 
+ 3*x0*x1*y0*y1 \
- 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 - 4*x1*x2*y1^2 \
+ 5*x0*x2*y0*y2 \
+ 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: T = PP(1, 1, 0, 1, 0, 0)
sage: X.Sxa(T[0])
Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
    y0 + y1,
    5*y0^2 + 7*y0*y1 + y1^2 + 11*y1*y2 + y2^2
```

$\text{Syb}(b)$

Function will return fiber by $S_b^y$.

This function is defined by:

$S_b^y = L_b^y \cap Q_b^y$.

Notation and definition from: [CS1996]

INPUT: b - Point in $\mathbb{P}^2$

OUTPUT: A subscheme representing the fiber

EXAMPLES:

```
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 
+ 3*x0*x1*y0*y1 \
- 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 - 4*x1*x2*y1^2 \
+ 5*x0*x2*y0*y2 \
+ 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: T = PP(1, 1, 0, 1, 0, 0)
sage: X.Syb(T[1])
Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
    x0,
    x0^2 + 3*x0*x1 + x1^2
```

canonical_height ($P, N, \text{badprimes}=\text{None}, \text{prec}=100$)

Evaluates the canonical height for $P$ with $N$ terms of the series of the local heights.

ALGORITHM:

The sum of the canonical height minus and canonical height plus, for more info see section 4 of [CS1996].

INPUT:

- $P$ – a surface point
- $N$ – positive integer (number of terms of the series to use)
• **badprimes** – (optional) list of integer primes (where the surface is degenerate)

• **prec** – (default: 100) float point or p-adic precision

**OUTPUT:** A real number

**EXAMPLES:**

```python
sage: set_verbose(None)
sage: R.<x0,x1,x2,y0,y1,y2> = PolynomialRing(QQ, 6)
sage: L = (-y0 - y1)*x0 + (-y0*x1 - y2*x2)
sage: Q = ((-y0^2 - y2*y0 - y2^2)*x1^2 + (-y2*y0 - y1^2)*x2)*x0 + ((-y0^2 - y2*y0 - y2^2)*x1^2 + (-y2*y0 - y1^2)*x2)*x1

sage: X = WehlerK3Surface([L, Q])
sage: P = X([1, 0, -1, 1, -1, 0])
# order 16
sage: X.canonical_height(P, 5)  # long time
0.00000000000000000000000000000
```

Call-Silverman example:

```python
call: set_verbose(None)
call: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
call: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 3*x0*x1*y0*y1

+ 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 - 4*x1*x2*y1^2 +

-5*x0*x2*y0*y2
-4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2

sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: P = X(0, 1, 0, 0, 0, 1)
sage: X.canonical_height(P, 4)
0.69826458668659859569990618895
```

**canonical_height_minus** \((P, N, badprimes=None, prec=100)\)

Evaluates the canonical height minus function of Call-Silverman for \(P\) with \(N\) terms of the series of the local heights.

Must be over \(\mathbb{Z}\) or \(\mathbb{Q}\).

**ALGORITHM:**

Sum over the lambda minus heights (local heights) in a convergent series, for more detail see section 7 of [CS1996].

**INPUT:**

• \(P\) – a surface point

• \(N\) – positive integer (number of terms of the series to use)

• **badprimes** – (optional) list of integer primes (where the surface is degenerate)

• **prec** – (default: 100) float point or p-adic precision

**OUTPUT:** A real number

**EXAMPLES:**

```python
call: set_verbose(None)
call: R.<x0,x1,x2,y0,y1,y2> = PolynomialRing(QQ, 6)
```
Call-Silverman example:

```python
sage: set_verbose(None)
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 +
   3*x0*x1*y0*y1 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 -
   4*x1*x2*y1^2 + 5*x0*x2*y0*y2 - 2*x2^2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 +
   x0*x1*y2^2 + 3*x2^2*y2^2
sage: X = WehlerK3Surface([Z, Y])
sage: P = X([0, 1, 0, 0, 0, 1])
sage: X.canonical_height_minus(P, 4)  # long time
0.55073705369676788175590206734
```

**canonical_height_plus** \((P, N, \text{badprimes}=None, \text{prec}=100)\)

Evaluates the canonical height plus function of Call-Silverman for \(P\) with \(N\) terms of the series of the local heights.

Must be over \(\mathbb{Z}\) or \(\mathbb{Q}\).

**ALGORITHM:**

Sum over the lambda plus heights (local heights) in a convergent series, for more detail see section 7 of [CS1996].

**INPUT:**

- \(P\) – a surface point
- \(N\) – positive integer. Number of terms of the series to use
- \(\text{badprimes}\) – (optional) list of integer primes (where the surface is degenerate)
- \(\text{prec}\) – (default: 100) float point or p-adic precision

**OUTPUT:** A real number

**EXAMPLES:**

```python
sage: set_verbose(None)
sage: R.<x0,x1,x2,y0,y1,y2> = PolynomialRing(QQ, 6)
sage: L = (-y0 - y1)*x0 + (-y0*x1 - y2*x2)
sage: Q = (-y2*y0 - y1^2)*x0^2 + ((-y0^2 - y2*y0 + (-y2*y1 - y2^2))*x1 +
   (-y0^2 - y2*y1)*x2)*x0 + ((-y0^2 - y2*y0 - y2^2)*x1^2 + (-y2*y0 - y1^2)*x2*x1,
   + (-y0^2 + (-y1 - y2)*y0)*x2^2)
sage: X = WehlerK3Surface([L, Q])
sage: P = X([1, 0, -1, 1, -1, 0])  # order 16
```

(continues on next page)
Call-Silverman Example:

```python
sage: set_verbose(None)
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 
3*x0*x1*y0*y1 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 
- 4*x1*x2*y1^2 + 5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 
+ x0*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: P = X([0, 1, 0, 0, 0, 1])
sage: X.canonical_height_plus(P, 4)  # long time
0.14752753298983071394400412161
```

**change_ring** (*R*)

Changes the base ring on which the Wehler K3 Surface is defined.

**INPUT:** *R* - ring

**OUTPUT:** K3 Surface defined over input ring

**EXAMPLES:**

```python
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], GF(3))
sage: L = x0*y0 + x1*y1 - x2*y2
sage: Q = x0*x1*y1^2 + x2^2*y0*y2
sage: W = WehlerK3Surface([L, Q])
sage: W.base_ring()
Finite Field of size 3
sage: T = W.change_ring(GF(7))
sage: T.base_ring()
Finite Field of size 7
```

**degenerate_fibers**

Function will return the (rational) degenerate fibers of the surface defined over the base ring, or the fraction field of the base ring if it is not a field.

**ALGORITHM:**

The criteria for degeneracy by the common vanishing of the polynomials `self.Gpoly(1, 0), self.Gpoly(1, 1), self.Gpoly(1, 2), self.Hpoly(1, 0, 1), self.Hpoly(1, 0, 2), self.Hpoly(1, 1, 2)` (for the first component), is from Proposition 1.4 in the following article: [CS1996].

This function finds the common solution through elimination via Groebner bases by using the .variety() function on the three affine charts in each component.

**OUTPUT:** The output is a list of lists where the elements of lists are points in the appropriate projective space. The first list is the points whose pullback by the projection to the first component (projective space) is dimension greater than 0. The second list is points in the second component

**EXAMPLES:**

```python
sage: R.<x0,x1,x2,y0,y1,y2> = PolynomialRing(ZZ, 6)
sage: Y = x0*y0 + x1*y1 - x2*y2
sage: Z = x0^2*y0*y1 + x0^2*y2^2 - x0*x1*y1*y2 + x1^2*y2*y1 + x2^2*y2^2
+ x0^2*y1^2 + x1^2*y2^2
```
sage: X = WehlerK3Surface([Z, Y])
sage: X.degenerate_fibers()
[[], [(1 : 0 : 0)]]

sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y1^2 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - 4*x0*x2*y1^2 + 5*x0*x2*y0*y2
- 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: X.degenerate_fibers()
[[], []]

sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: R = PP.coordinate_ring()
sage: l = y0*x0 + y1*x1 + (y0 - y1)*x2
sage: q = (y1*y0 + y2^2)*x0^2 + ((y0^2 - y2*y1)*x1 + (y0^2 + (y1^2 - y2^2))*x2)*x0 + ((y2*y0 + y1^2)*x1^2 + ((y2 + 2*y1)*y0 + (y1^2 + y2*y1 + 2*y2^2))*x2*x1
sage: X = WehlerK3Surface([l,q])
sage: X.degenerate_fibers()
[[(-1 : 1 : 1), (0 : 0 : 1)], [(-1 : -1 : 1), (0 : 0 : 1)]]

degenerate_primes (check=True)

Determine which primes $p$ self has degenerate fibers over $GF(p)$.

If check is False, then may return primes that do not have degenerate fibers. Raises an error if the surface is degenerate. Works only for ZZ or QQ.

INPUT: check – (default: True) boolean, whether the primes are verified

ALGORITHM:

$p$ is a prime of bad reduction if and only if the defining polynomials of self plus the G and H polynomials have a common zero. Or stated another way, $p$ is a prime of bad reduction if and only if the radical of the ideal defined by the defining polynomials of self plus the G and H polynomials is not $(x_0, x_1, \ldots, x_N)$.

This happens if and only if some power of each $x_i$ is not in the ideal defined by the defining polynomials of self (with G and H). This last condition is what is checked. The lcm of the coefficients of the monomials $x_i$ in a groebner basis is computed. This may return extra primes.

OUTPUT: List of primes.

EXAMPLES:

sage: R.<x0,x1,x2,y0,y1,y2> = PolynomialRing(QQ, 6)
sage: L = y0*x0 + (y1*x1 + y2*x2)
sage: Q = (2*y0^2 + y2*y0 + (2*y1^2 + y2^2))*x0^2 + ((y0^2 + y1*y0 + y1^2 + 2*y2*y1 + 2*y2^2)*x2)*x0 + ((2*y0^2 + y1^2 + 2*y2*y1 + 2*y2^2)*x2*x1
+ (y1 + 2*y2)*y0 + (2*y1^2 + y2*y1))*x1^2 + ((2*y1 + 2*y2)*y0 + (y1^2 + y2*y1 + 2*y2^2))*x2^2
sage: X = WehlerK3Surface([L, Q])
sage: X.degenerate_primes()
[[2, 3, 5, 11, 23, 47, 48747691, 111301831]]

fiber ($p$, component)

Returns the fibers $[y \ (\text{component} = 1) \text{ or } x \ (\text{Component} = 0)]$ of a point on a K3 Surface, will work for nondegenerate fibers only.
For algorithm, see [Hutz2007].

INPUT:
- \( p \) - a point in \( \mathbb{P}^2 \)

OUTPUT: The corresponding fiber (as a list)

EXAMPLES:

```python
sage: R.<x0,x1,x2,y0,y1,y2> = PolynomialRing(ZZ, 6)
sage: Y = x0*y0 + x1*y1 - x2*y2
sage: Z = y0^2*x0*x1 + y0^2*x2^2 - y0*y1*x1*x2 + y1^2*x2*x1 + y2^2*x2^2 +
     y2^2*x1^2 + y1^2*x2^2
sage: X = WehlerK3Surface([Z, Y])
sage: Proj = ProjectiveSpace(QQ, 2)
sage: P = Proj([1, 0, 0])
sage: X.fiber(P, 1)
Traceback (most recent call last):
  ...  
TypeError: fiber is degenerate
```

```python
sage: P.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 3*x0*x1*y0*y1,
     ...
    2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 - 4*x1*x2*y1^2 +
    5*x0*x2*y0*y2 - 7*x0^2*y1*y2 + 4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: Proj = P[0]
sage: T = Proj([0, 0, 1])
sage: X.fiber(T, 1)
\[(0 : 0 : 1 , 0 : 1 : 0), (0 : 0 : 1 , 2 : 0 : 0)\]
```

```python
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], GF(7))
sage: L = x0*y0 + x1*y1 - x2*y2
sage: Q = (2*x0^2 + x2*x0 + (2*x1^2 + x2^2))*y0^2 + ((x0^2 + x1*x0 +(x1^2 +
     ...
    2*x2^2)*x1 + (x1^2 + x2^2))*y1 + 
    (2*x1^2 + x2*x1 + x2^2)*y2)*y0 + ((2*x0^2+ (x1 + 2*x2)*x0 + (2*x1^2 +
     ...
    2*x2^2)*x1 + (x1^2 + x2^2))*y2*y1 + (2*x0^2 + x1*x0 + (2*x1^2 + x2^2))*y2^2)
```

```python
sage: W = WehlerK3Surface([L, Q])
sage: W.fiber([4, 0, 1], 0)
\[(0 : 1 : 0 , 4 : 0 : 1), (4 : 0 : 2 , 4 : 0 : 1)\]
```

`is_degenerate()`

Function will return True if there is a fiber (over the algebraic closure of the base ring) of dimension greater than 0 and False otherwise.

OUTPUT: boolean

EXAMPLES:

```python
sage: R.<x0,x1,x2,y0,y1,y2> = PolynomialRing(ZZ, 6)
sage: Y = x0*y0 + x1*y1 - x2*y2
sage: Z = x0^2*y0*y1 + x0^2*y2^2 - x0*x1*y1*y2 + x1^2*y2*y1 + x2^2*y2^2 +
     x2^2*y1^2 + x1^2*y2^2
sage: X = WehlerK3Surface([Z, Y])
```

(continues on next page)
sage: X.is_degenerate()  
True

sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)  
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 3*x0*x1*y0*y1  
    - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 - 4*x1*x2*y1^2 +  
    5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2  
sage: Y = x0*y0 + x1*y1 + x2*y2  
sage: X = WehlerK3Surface([Z, Y])  
sage: X.is_degenerate()  
False

sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], GF(3))  
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 3*x0*x1*y0*y1  
    - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 - 4*x1*x2*y1^2 +  
    5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2  
sage: Y = x0*y0 + x1*y1 + x2*y2  
sage: X = WehlerK3Surface([Z, Y])  
sage: X.is_degenerate()  
True

**is_isomorphic**(right)

Checks to see if two K3 surfaces have the same defining ideal.

**INPUT:**

- right - the K3 surface to compare to the original

**OUTPUT:** Boolean

**EXAMPLES:**

sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)  
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 3*x0*x1*y0*y1  
    - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 - 4*x1*x2*y1^2 +  
    5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2  
sage: Y = x0*y0 + x1*y1 + x2*y2  
sage: X = WehlerK3Surface([Z, Y])  
sage: W = WehlerK3Surface([Z + Y^2, Y])  
sage: X.is_isomorphic(W)  
True

sage: R.<x,y,z,u,v,w> = PolynomialRing(QQ, 6)  
sage: L = x*u-y*v  
sage: Q = x*y+v^2 + z^2+u*w  
sage: W1 = WehlerK3Surface([L, Q])  
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)  
sage: L = x0*y0 + x1*y1 + x2*y2  
sage: Q = x1^2*y0^2 + 2*x2^2*y0*y1 + x0^2*y1^2 -x0*x1*y2^2  
sage: N2 = WehlerK3Surface([L, Q])  
sage: W1.is_isomorphic(N2)  
False
**is_smooth()**
Function will return the status of the smoothness of the surface.

**ALGORITHM:**
Checks to confirm that all of the 2x2 minors of the Jacobian generated from the biquadratic and bilinear forms have no common vanishing points.

**OUTPUT:** Boolean

**EXAMPLES:**

```
sage: R.<x0,x1,x2,y0,y1,y2> = PolynomialRing(ZZ, 6)
sage: Y = x0*y0 + x1*y1 - x2*y2
sage: Z = x0^2*y0*y2 + x0^2*y2^2 - x0*x1*y1*y2 + x1^2*y2^2 + x2^2*y1^2 + x1^2*y2^2
sage: X = WehlerK3Surface([Z, Y])
sage: X.is_smooth()
False
```

```
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: L = x0*y0 + x1*y1 + x2*y2
sage: Q = x1^2*y0^2 + 2*x2^2*y0*y1 + x0^2*y1^2 - x0*x1*y2^2
sage: W = WehlerK3Surface([L, Q])
sage: T = W([-1, -1, 1, 1, 0, 1])
sage: X.is_symmetric_orbit(orbit)
True
```

**is_symmetric_orbit**(orbit)
Checks to see if the orbit is symmetric (i.e. if one of the points on the orbit is fixed by ‘sigma_x’ or ‘sigma_y’).

**INPUT:**
• orbit - a periodic cycle of either psi or phi

**OUTPUT:** Boolean

**EXAMPLES:**

```
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], GF(7))
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 3*x0*x1*y0*y1
     - x0^2*y1^2 + 2*x1*x2*y1^2 + 2*x2^2*y1^2 - x0*x2*y1^2 + 3*x0*x1*y0*y1 + \n- 4*x1*x2*y1^2 + 5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0*y2^2 + 4*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: T = PP([0, 0, 1, 1, 0, 0])
sage: orbit = X.orbit_psi(T, 4)
sage: X.is_symmetric_orbit(orbit)
True
```

(continues on next page)
lambda_minus \( (P, v, N, m, n, \text{prec}=100) \)
Evaluates the local canonical height minus function of Call-Silverman at the place \( v \) for \( P \) with \( N \) terms of the series.

Use \( v = 0 \) for the Archimedean place. Must be over \( \mathbb{Z} \) or \( \mathbb{Q} \).

ALGORITHM:
Sum over local heights using convergent series, for more details, see section 4 of [CS1996].

INPUT:
- \( P \) – a projective point
- \( N \) – positive integer. number of terms of the series to use
- \( v \) – non-negative integer. a place, use \( v = 0 \) for the Archimedean place
- \( m, n \) – positive integers, We compute the local height for the divisor \( E_{mn}^+ \). These must be indices of non-zero coordinates of the point \( P \).
- \( \text{prec} \) – (default: 100) float point or p-adic precision

OUTPUT: A real number

EXAMPLES:
```python
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 3*x0*x1*y0*y1 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - 5*x0*x2*y1^2 + y0^2 + x0^2*y0^2 + 7*x0^2*y1*y2 + 3*x1^2*y1^2 - 4*x1*x2*y1^2 + 4*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: P = X([0, 0, 1, 0, 0, 0])
sage: X.lambda_minus(P, 20, 2, 0, 200)
-0.18573351672047135037172805779671791488351056677474271893705
```

lambda_plus \( (P, v, N, m, n, \text{prec}=100) \)
Evaluates the local canonical height plus function of Call-Silverman at the place \( v \) for \( P \) with \( N \) terms of the series.

Use \( v = 0 \) for the archimedean place. Must be over \( \mathbb{Z} \) or \( \mathbb{Q} \).

ALGORITHM:
Sum over local heights using convergent series, for more details, see section 4 of [CS1996].

INPUT:
- \( P \) – a surface point
- \( N \) – positive integer. number of terms of the series to use
- \( v \) – non-negative integer. a place, use \( v = 0 \) for the Archimedean place
- \( m, n \) – positive integers, We compute the local height for the divisor \( E_{mn}^+ \). These must be indices of non-zero coordinates of the point \( P \).
• \texttt{prec} – (default: 100) float point or p-adic precision

OUTPUT: A real number

EXAMPLES:

```sage
PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 3*x0*x1*y0*y1 - 2*x0^2*y0^2 + x1^2*y1^2 + 2*x0^2*y1^2 - x0^2*x1*y1^2 - 4*x1*x2*y1^2 + 5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2
Y = x0*y0 + x1*y1 + x2*y2
X = WehlerK3Surface([Z, Y])
P = X([0, 0, 1, 1, 0, 0])
P = X.lambda_plus(P, 0, 10, 2, 0)
0.89230705169161608922595928129
```

\texttt{nth_iterate_phi} \((P, n, **kwds)\)

Computes the nth iterate for the phi function.

INPUT:

• \(P\) – a point in \(\mathbb{P}^2 \times \mathbb{P}^2\)
• \(n\) – an integer

kwds:

• \texttt{check} - (default: True) boolean checks to see if point is on the surface
• \texttt{normalize} – (default: False) boolean normalizes the point

OUTPUT: The nth iterate of the point given the phi function (if \(n\) is positive), or the psi function (if \(n\) is negative)

EXAMPLES:

```sage
PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
L = x0*y0 + x1*y1 + x2*y2
Q = x1^2*y0^2 + 2*x2^2*y0*y1 + x0^2*y1^2 - x0*x1*y2^2
W = WehlerK3Surface([L, Q])
T = W([-1, -1, 1, 1, 0, 1])
W.nth_iterate_phi(T, 7)
(-1 : 0 : 1 , 1 : -2 : 1)
```

```sage
PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
L = x0*y0 + x1*y1 + x2*y2
Q = x1^2*y0^2 + 2*x2^2*y0*y1 + x0^2*y1^2 - x0*x1*y2^2
W = WehlerK3Surface([L, Q])
T = W([-1, -1, 1, 1, 0, 1])
W.nth_iterate_phi(T, -7)
(1 : 0 : 1 , -1 : 2 : 1)
```

```sage
R.<x0,x1,x2,y0,y1,y2>=PolynomialRing(QQ, 6)
L = (−y0 − y1)*x0 + (−y0*x1 − y2*x2)
Q = (−y2*y0 − y1^2)*x0^2 + ((−y0^2 − y2*y0 + (−y2*y1 − y1^2))*x1 + (−y0^2 − y2*y1)*x2)*x0 
+ ((−y0^2 − y2*y0 − y1^2)*x1^2 + (−y2*y0 − y1^2)*x2*x1 + (−y0^2 + (−y1 − y2*y1)*x2^2))
X = WehlerK3Surface([L, Q])
P = X([1, 0, -1, 1, -1, 0])
```

(continues on next page)
\texttt{sage}: \texttt{X.nth_iterate_phi(P, 8) == X.nth_iterate_psi(P, 8)}

True

\textbf{nth_iterate_psi} \((P, n, **kwds)\)

Computes the nth iterate for the psi function.

\textbf{INPUT}:

\begin{itemize}
  \item \texttt{P} - a point in \(\mathbb{P}^2 \times \mathbb{P}^2\)
  \item \texttt{n} - an integer
\end{itemize}

\textbf{kwds}:

\begin{itemize}
  \item \texttt{check} - (default: True) boolean, checks to see if point is on the surface
  \item \texttt{normalize} - (default: False) boolean, normalizes the point
\end{itemize}

\textbf{OUTPUT}: The nth iterate of the point given the psi function (if \texttt{n} is positive), or the phi function (if \texttt{n} is negative)

\textbf{EXAMPLES}:

\begin{verbatim}
\texttt{sage}: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
\texttt{sage}: L = x0*y0 + x1*y1 + x2*y2
\texttt{sage}: Q = x1^2*y0^2 + 2*x2^2*y0*y1 + x0^2*y1^2 - x0*x1*y2^2
\texttt{sage}: W = WehlerK3Surface([L, Q])
\texttt{sage}: T = W([-1, -1, 1, 1, 0, 1])
\texttt{sage}: W.nth_iterate_psi(T, -7)
\end{verbatim}

\((-1 : 0 : 1 , 1 : -2 : 1)\)

\begin{verbatim}
\texttt{sage}: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
\texttt{sage}: L = x0*y0 + x1*y1 + x2*y2
\texttt{sage}: Q = x1^2*y0^2 + 2*x2^2*y0*y1 + x0^2*y1^2 - x0*x1*y2^2
\texttt{sage}: W = WehlerK3Surface([L, Q])
\texttt{sage}: T = W([-1, -1, 1, 1, 0, 1])
\texttt{sage}: W.nth_iterate_psi(T, 7)
\end{verbatim}

\((1 : 0 : 1 , -1 : 2 : 1)\)

\textbf{orbit PHI} \((P, N, **kwds)\)

Returns the orbit of the \(\phi\) function defined by \(\phi = \sigma_y \circ \sigma_x\) Function is defined in [CS1996].

\textbf{INPUT}:

\begin{itemize}
  \item \texttt{P} - Point on the K3 surface
  \item \texttt{N} - a non-negative integer or list or tuple of two non-negative integers
\end{itemize}

\textbf{kwds}:

\begin{itemize}
  \item \texttt{check} - (default: True) boolean, checks to see if point is on the surface
  \item \texttt{normalize} - (default: False) boolean, normalizes the point
\end{itemize}

\textbf{OUTPUT}: List of points in the orbit

\textbf{EXAMPLES}:

\begin{verbatim}
\texttt{sage}: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
\texttt{sage}: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + \
3*x0*x1*y0*y1 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 - \
4*x1*x2*y1^2 + 5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 + \
\end{verbatim}

(continues on next page)
\[ x_0 \cdot x_1 \cdot y_2^2 + 3 \cdot x_2^2 \cdot y_2^2 \]

```
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: T = PP(0, 0, 1, 1, 0, 0)
sage: X.orbit_phi(T, 2, normalize = True)
[(0 : 0 : 1 , 1 : 0 : 0), (-1 : 0 : 1 , 0 : 1 : 0), (-12816/6659 : 55413/6659 : 1, 1 : 1/9 : 1)]
sage: X.orbit_phi(T, [2, 3], normalize = True)
[(-12816/6659 : 55413/6659 : 1 , 1 : 1/9 : 1),
  (-117756062505511/54767410965117 : -2313404798794359/37466994368025041 : -1)]
```

\( \text{orbit}_\psi(P, N, **\text{kwd}) \)

Returns the orbit of the \( \psi \) function defined by \( \psi = \sigma_x \circ \sigma_y \).

Function is defined in [CS1996].

**INPUT:**

- \( P \) - a point on the K3 surface
- \( N \) - a non-negative integer or list or tuple of two non-negative integers

**kwds:**

- \( \text{check} \) - (default: True) boolean, checks to see if point is on the surface
- \( \text{normalize} \) - (default: False) boolean, normalizes the point

**OUTPUT:** a list of points in the orbit

**EXAMPLES:**

```
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 
    3*x0*x1*y0*y1 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 - 
    4*x1*x2*y1^2 + 5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 + 
    x0*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: T = X(0, 0, 1, 1, 0, 0)
sage: X.orbit_psi(T, 2, normalize = True)
[(0 : 0 : 1 , 1 : 0 : 0), (0 : 0 : 1 , 0 : 1 : 0), (-1 : 0 : 1 , 1 : 1/9 : 1)]
sage: X.orbit_psi(T, [2, 3], normalize = True)
[(-1 : 0 : 1 , 1 : 1/9 : 1),
  (-12816/6659 : 55413/6659 : 1 , -117756062505511/54767410965117 : -2313404798794359/37466994368025041 : -1)]
```

\( \phi(a, **\text{kwd}) \)

Evaluates the function \( \phi = \sigma_y \circ \sigma_x \).

**ALGORITHM:**

Refer to Section 6: “An algorithm to compute \( \sigma_x, \sigma_y, \phi \), and \( \psi \)” in [CS1996].

For the degenerate case refer to [FH2015].

**INPUT:**
• a - Point in \( \mathbb{P}^2 \times \mathbb{P}^2 \)

kwds:
• check - (default: True) boolean checks to see if point is on the surface
• normalize - (default: True) boolean normalizes the point

OUTPUT: A point on this surface

EXAMPLES:

```
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + \
    3*x0*x1*y0*y1 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 \n    - 4*x1*x2*y1^2 + 5*x0*x2*y0*y2 -4*x1*x2*y0*y1 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 \n    + x0*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: T = PP([0, 0, 1, 1, 0, 0])
sage: X.phi(T)
(-1 : 0 : 1 , 0 : 1 : 0)
```

\( \psi \) (a, **kwds)

Evaluates the function \( \psi = \sigma_x \circ \sigma_y \).

ALGORITHM:

Refer to Section 6: “An algorithm to compute \( \sigma_x, \sigma_y, \phi, \) and \( \psi \)” in [CS1996].

For the degenerate case refer to [FH2015].

INPUT:
• a - Point in \( \mathbb{P}^2 \times \mathbb{P}^2 \)

kwds:
• check - (default: True) boolean checks to see if point is on the surface
• normalize - (default: True) boolean normalizes the point

OUTPUT: A point on this surface

EXAMPLES:

```
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + \
    3*x0*x1*y0*y1 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 \n    - 4*x1*x2*y1^2 + 5*x0*x2*y0*y2 -4*x1*x2*y0*y1 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 \n    + x0*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: T = PP([0, 0, 1, 1, 0, 0])
sage: X.psi(T)
(0 : 0 : 1 , 0 : 1 : 0)
```

\( \sigma_{X} \) (P, **kwds)

Function returns the involution on the Wehler K3 surface induced by the double covers.

In particular, it fixes the projection to the first coordinate and swaps the two points in the fiber, i.e. \( (x, y) \rightarrow (x, y') \). Note that in the degenerate case, while we can split fiber into pairs of points, it is not always possible to distinguish them, using this algorithm.
ALGORITHM:
Refer to Section 6: “An algorithm to compute $\sigma_x, \sigma_y, \phi,$ and $\psi$” in [CS1996FH2015. For the degenerate case refer to [FH2015].

INPUT:
- $P$ - a point in $\mathbb{P}^2 \times \mathbb{P}^2$

kwds:
- $\text{check}$ - (default: True) boolean checks to see if point is on the surface
- $\text{normalize}$ – (default: True) boolean normalizes the point

OUTPUT: A point on the K3 surface

EXAMPLES:

```sage
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + \
3*x0*x1*y0*y1 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 - \
4*x1*x2*y1^2 + 5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + \
4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: T = PP(0, 0, 1, 1, 0, 0)
sage: X.sigmaX(T)
(0 : 0 : 1 , 0 : 1 : 0)
```

degenerate examples:

```sage
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: l = y0*x0 + y1*x1 + (y0 - y1)*x2
sage: q=-3*x0^2*y0^2 + 4*x0*x1*y0^2 - 3*x0*x2*y0^2 - 5*x0^2*y0*y1 - \
190*x0*x1*y0*y1 - 5*x1^2*y0*y1 + 5*x0*x2*y0*y1 + 14*x1*x2*y0*y1 + \
5*x2^2*y0*y1 - x0^2*y1^2 - 6*x0*x1*y1^2 - 2*x1^2*y1^2 - 2*x0*x2*y1^2 - \
4*x2^2*y1^2 + 4*x0^2*y0*y2 - x1^2*y0*y2 + 3*x0*x2*y0*y2 + 6*x1*x2*y0*y2 - \
6*x0^2*y1*y2 - 4*x0*x1*y1*y2 - x1^2*y1*y2 + 51*x0*x2*y1*y2 - 7*x1*x2*y1*y2 - \
9*x2^2*y1*y2 - x0^2*y2^2 - 4*x0*x1*y2^2 + 4*x1^2*y2^2 - x0*x2*y2^2 - \
13*x1*x2*y2^2 - x2^2*y2^2
sage: X = WehlerK3Surface([l, q])
sage: P = X([1, 0, 0, 1, 0, 1])
```

Case where we cannot distinguish the two points:

```sage
sage: PP.<y0,y1,y2,x0,x1,x2>=ProductProjectiveSpaces([2, 2], GF(3))
sage: l = x0*y0 + x1*y1 + x2*y2
sage: q=-3*x0^2*y0^2 + 4*x0*x1*y0^2 - 3*x0*x2*y0^2 - 5*x0^2*y0*y1 - \
190*x0*x1*y0*y1 - 5*x1^2*y0*y1 + 5*x0*x2*y0*y1 + 14*x1*x2*y0*y1 + \
5*x2^2*y0*y1 - x0^2*y1^2 - 6*x0*x1*y1^2 - 2*x1^2*y1^2 - 2*x0*x2*y1^2 - \
4*x2^2*y1^2 + 4*x0^2*y0*y2 - x1^2*y0*y2 + 3*x0*x2*y0*y2 + 6*x1*x2*y0*y2 - \
6*x0^2*y1*y2 - 4*x0*x1*y1*y2 - x1^2*y1*y2 + 51*x0*x2*y1*y2 - 7*x1*x2*y1*y2 - \
9*x2^2*y1*y2 - x0^2*y2^2 - 4*x0*x1*y2^2 + 4*x1^2*y2^2 - x0*x2*y2^2 - \
13*x1*x2*y2^2 - x2^2*y2^2
sage: X = WehlerK3Surface([l, q])
sage: P = X([1, 0, 1, 0, 1, 1])
```

(continues on next page)
`sage: X.sigmaX(X.sigmaX(P))`
Traceback (most recent call last):
...
ValueError: cannot distinguish points in the degenerate fiber

`sigmaY(P, **kwds)`
Function returns the involution on the Wehler K3 surfaces induced by the double covers.

In particular, it fixes the projection to the second coordinate and swaps the two points in the fiber, i.e. $(x, y) \rightarrow (x', y)$. Note that in the degenerate case, while we can split the fiber into two points, it is not always possible to distinguish them, using this algorithm.

ALGORITHM:
Refer to Section 6: “An algorithm to compute $\sigma_x, \sigma_y, \phi$, and $\psi$” in [CS1996]. For the degenerate case refer to [FH2015].

INPUT:

- $P$ - a point in $\mathbb{P}^2 \times \mathbb{P}^2$

kwds:

- `check` - (default: True) boolean checks to see if point is on the surface
- `normalize` - (default: True) boolean normalizes the point

OUTPUT: A point on the K3 surface

EXAMPLES:

```sage
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 + 3*x0*x1*y0*y1 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 - 4*x1*x2*y1^2 + 5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: T = PP(0, 0, 1, 1, 0, 0)
sage: X.sigmaY(T)
(0 : 0 : 1 , 1 : 0 : 0)
```

degenerate examples:

```sage
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: l = y0*x0 + y1*x1 + (y0 - y1)*x2
sage: q = (y1*y0)*x0^2 + ((y0^2)*x1 + (y0^2 + (y1^2 - y2^2))*x2)*x0 + (y2*y0 + y1^2)*x1^2 + (y0^2 + (-y1^2 + y2^2))*x2*x1
sage: X = WehlerK3Surface([l, q])
sage: X.sigmaY(X([1, -1, 0, -1, -1, 1]))
(1/10 : -1/10 : 1 , -1 : -1 : 1)
sage: X.sigmaY(X([0, 0, 1, -1, -1, 1]))
(-4 : 4 : 1 , -1 : -1 : 1)
sage: X.sigmaY(X([1, 2, 0, 0, 0, 1]))
(-3 : -3 : 1 , 0 : 0 : 1)
sage: X.sigmaY(X([1, 1, 1, 0, 0, 1]))
(1 : 0 : 0 , 0 : 0 : 1)
```

Case where we cannot distinguish the two points:
sage: PP.<x0,x1,x2,y0,y1,y2>=ProductProjectiveSpaces([2, 2], GF(3))
sage: l = x0*y0 + x1*y1 + x2*y2
sage: q=-3*x0^2*y0^2 + 4*x0*x1*y0^2 - 3*x0*x2*y0^2 - 5*x0^2*y0*y1 -
\rightarrow 190*x0*x1*y0*y1 \
- 5*x1^2*y0*y1 + 5*x0*x2*y0*y1 + 14*x1*x2*y0*y1 + 5*x2^2*y0*y1 - x0^2*y1^2 -
\rightarrow 6*x0*x1*y1^2 \
- 2*x1^2*y1^2 + 2*x0*x2*y1^2 - 4*x2^2*y1^2 + 4*x0^2*y0*y2 - x1^2*y0*y2 +
\rightarrow 3*x0*x2*y0*y2 \
+ 6*x1*x2*y0*y2 - 6*x0^2*y1*y2 - 4*x0*x1*y1*y2 - x1^2*y1*y2 + 51*x0*x2*y1*y2 -
\rightarrow 7*x1*x2*y1*y2 \
- 9*x2^2*y1*y2 - x0^2*y2^2 - 4*x0*x1*y2^2 + 4*x1^2*y2^2 - x0*x2*y2^2 +
\rightarrow 13*x1*x2*y2^2 - x2^2*y2^2
sage: X = WehlerK3Surface([l ,q])
sage: P = X([0, 1, 1, 1, 0, 0])
sage: X.sigmaY(X.sigmaY(P))
Traceback (most recent call last):
 ... ValueError: cannot distinguish points in the degenerate fiber

sage.dynamics.arithmetic_dynamics.wehlerK3.random_WehlerK3Surface(PP)
Produces a random K3 surface in $\mathbb{P}^2 \times \mathbb{P}^2$ defined as the intersection of a bilinear and biquadratic form. [Weh1998]

INPUT: Projective space cartesian product

OUTPUT: WehlerK3Surface_ring

EXAMPLES:

sage: PP.<x0, x1, x2, y0, y1, y2> = ProductProjectiveSpaces([2, 2], GF(3))
sage: random_WehlerK3Surface(PP)

Closed subscheme of Product of projective spaces $\mathbb{P}^2 \times \mathbb{P}^2$ over Finite Field of size 3 defined by:
x0*y0 + x1*y1 + x2*y2,
-x1^2*y0^2 - x2^2*y0^2 + x0^2*y0*y1 - x0*x1*y0*y1 - x1^2*y0*y1 + x1*x2*y0*y1 + x0^2*y1^2 + x0*x1*y1^2 - x1^2*y1^2 + x0*x2*y1^2 - x0^2*y2^2 + x0*x1*y2^2 + x1*x2*y2^2 - x0*x2*y2^2

See also:

- sage.schemes.affine.affine_morphism
- sage.schemes.projective.projective_morphism
- sage.schemes.product_projective.morphism

6.5. Wehler K3 Surfaces 261
INDICES AND TABLES

- Index
- Module Index
- Search Page

d
sage.dynamics.arithmetic_dynamics.affine_ds, 177
sage.dynamics.arithmetic_dynamics.generic_ds, 171
sage.dynamics.arithmetic_dynamics.product_projective_ds, 239
sage.dynamics.arithmetic_dynamics.projective_ds, 187
sage.dynamics.arithmetic_dynamics.wehlerK3, 241
sage.dynamics.cellular_automata.solitons, 1
sage.dynamics.complex_dynamics.mandel_julia, 19
sage.dynamics.flat_surfaces.quadratic_strata, 40
sage.dynamics.flat_surfaces.strata, 25
sage.dynamics.interval_exchanges.constructors, 41
sage.dynamics.interval_exchanges.iet, 109
sage.dynamics.interval_exchanges.labelled, 51
sage.dynamics.interval_exchanges.reduced, 72
sage.dynamics.interval_exchanges.template, 82

S
sage.sandpiles.sandpile, 117
INDEX

A

AbelianStrata() (in module sage.dynamics.flat_surfaces.strata), 29
AbelianStrata_all (class in sage.dynamics.flat_surfaces.strata), 30
AbelianStrata_d (class in sage.dynamics.flat_surfaces.strata), 31
AbelianStrata_g (class in sage.dynamics.flat_surfaces.strata), 31
AbelianStrata_gd (class in sage.dynamics.flat_surfaces.strata), 31
AbelianStratum (class in sage.dynamics.flat_surfaces.strata), 31
add_random() (sage.sandpiles.sandpile.SandpileConfig method), 143
add_random() (sage.sandpiles.sandpile.SandpileDivisor method), 152
admissible_partitions() (in module sage.sandpiles.sandpile), 164
all_k_config() (sage.sandpiles.sandpile.Sandpile method), 122
all_k_div() (sage.sandpiles.sandpile.Sandpile method), 122
all_minimal_models() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 189
all_rational_preimages() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 225
alphabet() (sage.dynamics.interval_exchanges.template.Permutation method), 84
alphabet() (sage.dynamics.interval_exchanges.template.RauzyDiagram method), 103
alphabetized_atwin() (in module sage.dynamics.interval_exchanges.reduced), 81
alphabetized_qtwin() (in module sage.dynamics.interval_exchanges.reduced), 81
append() (sage.dynamics.interval_exchanges.template.RauzyDiagram.Path method), 101
arf_invariant() (sage.dynamics.interval_exchanges.template.PermutationIET method), 91
as_scheme_morphism() (sage.dynamics.arithmetic_dynamics.generic_ds.DynamicalSystem method), 173
attached_in_degree() (sage.dynamics.interval_exchanges.template.PermutationIET method), 92
attached_out_degree() (sage.dynamics.interval_exchanges.template.PermutationIET method), 92
attached_type() (sage.dynamics.interval_exchanges.template.PermutationIET method), 93
automorphism_group() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 190
automorphism_group() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_finite_field method), 235
avalanche_polynomial() (sage.sandpiles.sandpile.Sandpile method), 123
aztec_sandpile() (in module sage.sandpiles.sandpile), 165

B

betti() (sage.sandpiles.sandpile.Sandpile method), 123
betti() (sage.sandpiles.sandpile.SandpileDivisor method), 152
betti_complexes() (sage.sandpiles.sandpile.Sandpile method), 123
burning_config() (sage.sandpiles.sandpile.Sandpile method), 124
burning_script() (sage.sandpiles.sandpile.Sandpile method), 125
burst_size() (sage.sandpiles.sandpile.SandpileConfig method), 144

C

canonical_divisor() (sage.sandpiles.sandpile.Sandpile method), 126
canonical_height() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 192
canonical_height() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeplerK3Surface_ring method), 246
canonical_height_minus() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeplerK3Surface_ring method), 247
canonical_height_plus() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeplerK3Surface_ring method), 248
cardinality() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeplerK3Surface_finite_field method), 242
cardinality() (sage.dynamics.interval_exchanges.template.RauzyDiagram method), 103
CCA (in module sage.dynamics.flat_surfaces.strata), 34
change_ring() (sage.dynamics.arithmetic_dynamics.generic_ds.DynamicalSystem method), 173
change_ring() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeplerK3Surface_ring method), 249
complete() (sage.dynamics.interval_exchanges.template.FlippedRauzyDiagram method), 83
complete() (sage.dynamics.interval_exchanges.template.RauzyDiagram method), 104
composition() (sage.dynamics.interval_exchanges.template.RauzyDiagram.Path method), 101
conjugate() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 193
conjugating_set() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 226
connected_component() (sage.dynamics.interval_exchanges.template.PermutationIET method), 93
connected_components() (sage.dynamics.flat_surfaces.strata.AbelianStratum method), 32
connected_rational_component() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 227
ConnectedComponentOfAbelianStratum (class in sage.dynamics.flat_surfaces.strata), 34
critical_height() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 194
critical_point_portrait() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 195
critical_points() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 196
cyclegraph() (sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine_finite_field method), 186
cyclegraph() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_finite_field method), 237
cylindric() (sage.dynamics.interval_exchanges.template.PermutationIET method), 94

D

Dcomplex() (sage.sandpiles.sandpile.SandpileDivisor method), 151
decompose() (sage.dynamics.interval_exchanges.template.PermutationIET method), 95
deg() (sage.sandpiles.sandpile.SandpileConfig method), 145
deg() (sage.sandpiles.sandpile.SandpileDivisor method), 153
degenerate_fibers() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeplerK3Surface_ring method), 249
degenerate_primes() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeplerK3Surface_ring method), 250
degree_sequence() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 197
dehomogenizet() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 197
dict() (sage.sandpiles.sandpile.Sandpile method), 126
domain_singularities() (sage.dynamics.interval_exchanges.i.et.IntervalExchangeTransformation method), 111
dual_substitution() (sage.dynamics.interval_exchanges.labelled.LabelledRauzyDiagram.Path method), 67
dualize() (sage.sandpiles.sandpile.SandpileConfig method), 145
dualize() (sage.sandpiles.sandpile.SandpileDivisor method), 153
dynamical_degree() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 198
DynamicalSystem (class in sage.dynamics.arithmetic_dynamics.generic_ds), 171
DynamicalSystem_affine (class in sage.dynamics.arithmetic_dynamics.affine_ds), 177
DynamicalSystem_affine_field (class in sage.dynamics.arithmetic_dynamics.affine_ds), 185
DynamicalSystem_affine_finite_field (class in sage.dynamics.arithmetic_dynamics.affine_ds), 186
DynamicalSystem_product_projective (class in sage.dynamics.arithmetic_dynamics.product_projective_ds), 239
DynamicalSystem_projective (class in sage.dynamics.arithmetic_dynamics.projective_ds), 187
DynamicalSystem_projective_field (class in sage.dynamics.arithmetic_dynamics.projective_ds), 224
DynamicalSystem_projective_finite_field (class in sage.dynamics.arithmetic_dynamics.projective_ds), 235
dynatomic_polynomial() (sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine method), 179
dynatomic_polynomial() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 198

E
edge_iterator() (sage.dynamics.interval_exchanges.template.RauzyDiagram method), 104
dynatomic_polynomial() (sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine method), 179
dynatomic_polynomial() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 198

F
fiber() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 250
field_of_definition_critical() (sage.dynamics.arithmetic_dynamics.generic_ds.DynamicalSystem method), 174
field_of_definition_periodic() (sage.dynamics.arithmetic_dynamics.generic_ds.DynamicalSystem method), 175
field_of_definition_preimage() (sage.dynamics.arithmetic_dynamics.generic_ds.DynamicalSystem method), 176
fire_script() (sage.sandpiles.sandpile.SandpileDivisor method), 154
fire_script() (sage.sandpiles.sandpile.SandpileConfig method), 146
fire_unstable() (sage.sandpiles.sandpile.SandpileDivisor method), 147
fire_unstable() (sage.sandpiles.sandpile.SandpileConfig method), 147
fire_vertex() (sage.sandpiles.sandpile.SandpileDivisor method), 155
fire_vertex() (sage.sandpiles.sandpile.SandpileConfig method), 147
firing_graph() (in module sage.sandpiles.sandpile), 165
FlippedLabelledPermutation (class in sage.dynamics.interval_exchanges.labelled), 51
FlippedLabelledPermutationIET (class in sage.dynamics.interval_exchanges.labelled), 52
FlippedLabelledPermutationLI (class in sage.dynamics.interval_exchanges.labelled), 54
FlippedLabelledRauzyDiagram (class in sage.dynamics.interval_exchanges.labelled), 56
FlippedPermutation (class in sage.dynamics.interval_exchanges.template), 82
FlippedPermutationIET (class in sage.dynamics.interval_exchanges.template), 82
FlippedPermutationLI (class in sage.dynamics.interval_exchanges.template), 83
FlippedRauzyDiagram (class in sage.dynamics.interval_exchanges.template), 83
FlippedReducedPermutation (class in sage.dynamics.interval_exchanges.reduced), 72
FlippedReducedPermutationIET (class in sage.dynamics.interval_exchanges.reduced), 72
FlippedReducedPermutationLI (class in sage.dynamics.interval_exchanges.reduced), 72
flips() (sage.dynamics.interval_exchanges.template.FlippedPermutationIET method), 82
flips() (sage.dynamics.interval_exchanges.template.FlippedPermutationLI method), 83
full_loop_iterator() (sage.dynamics.interval_exchanges.labelled.LabelledRauzyDiagram method), 71
full_nloop_iterator() (sage.dynamics.interval_exchanges.labelled.LabelledRauzyDiagram method), 71

G

GeneralizedPermutation() (in module sage.dynamics.interval_exchanges.constructors), 44
genus() (sage.dynamics.flat_surfaces.quadratic_strata.QuadraticStratum method), 40
genus() (sage.dynamics.flat_surfaces.strata.AbelianStratum method), 33
genus() (sage.dynamics.flat_surfaces.strata.ConnectedComponentOfAbelianStratum method), 35
genus() (sage.dynamics.interval_exchanges.template.PermutationIET method), 95
glue_graphs() (in module sage.sandpiles.sandpile), 165
Gpoly() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeherK3Surface_ring method), 242
graph() (sage.dynamics.interval_exchanges.template.RauzyDiagram method), 107
green_function() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 202
groebner() (sage.sandpiles.sandpile.Sandpile method), 126
group_gens() (sage.sandpiles.sandpile.Sandpile method), 127
group_order() (sage.sandpiles.sandpile.Sandpile method), 127

H

h_vector() (sage.sandpiles.sandpile.Sandpile method), 127
has_rauzy_move() (sage.dynamics.interval_exchanges.labelled.LabelledPermutationIET method), 61
has_rauzy_move() (sage.dynamics.interval_exchanges.reduced.ReducedPermutationIET method), 77
has_rauzy_move() (sage.dynamics.interval_exchanges.template.Permutation method), 84
has_right_rauzy_move() (sage.dynamics.interval_exchanges.labelled.LabelledPermutationLI method), 64
has_right_rauzy_move() (sage.dynamics.interval_exchanges.template.PermutationLI method), 99
height_difference_bound() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 203
help() (sage.sandpiles.sandpile.Sandpile static method), 127
help() (sage.sandpiles.sandpile.SandpileConfig static method), 147
help() (sage.sandpiles.sandpile.SandpileDivisor static method), 155
hilbert_function() (sage.sandpiles.sandpile.Sandpile method), 129
homogenize() (sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine method), 181
horizontal_inverse() (sage.dynamics.interval_exchanges.template.Permutation method), 85
Hpoly() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeherK3Surface_ring method), 243
HypCCA (in module sage.dynamics.flat_surfaces.strata), 38
HypConnectedComponentOfAbelianStratum (class in sage.dynamics.flat_surfaces.strata), 38
Index
LabelledPermutation (class in sage.dynamics.interval_exchanges.labelled), 56
LabelledPermutationIET (class in sage.dynamics.interval_exchanges.labelled), 60
LabelledPermutationLI (class in sage.dynamics.interval_exchanges.labelled), 64
LabelledPermutationsIET_iterator() (in module sage.dynamics.interval_exchanges.labelled), 67
LabelledRauzyDiagram (class in sage.dynamics.interval_exchanges.labelled), 67
LabelledRauzyDiagram.Path (class in sage.dynamics.interval_exchanges.labelled), 67
lambda_minus() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 254
lambda_plus() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 254
laplacian() (sage.sandpiles.sandpile.Sandpile method), 132
latex_state_evolution() (sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata method), 11
latex_states() (sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata method), 11
left_rauzy_move() (sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutationLI method), 55
left_rauzy_move() (sage.dynamics.interval_exchanges.labelled.LabelledPermutationLI method), 65
left_right_inverse() (sage.dynamics.interval_exchanges.template.Permutation method), 86
length() (sage.dynamics.interval_exchanges.iet.IntervalExchangeTransformation method), 113
length() (sage.dynamics.interval_exchanges.labelled.LabelledPermutation method), 57
length() (sage.dynamics.interval_exchanges.reduced.ReducedPermutation method), 75
length() (sage.dynamics.interval_exchanges.reduced.ReducedPermutation method), 75
length() (sage.dynamics.interval_exchanges.reduced.ReducedPermutation method), 76
length() (sage.dynamics.interval_exchanges.reduced.ReducedPermutation method), 76
length() (sage.dynamics.interval_exchanges.reduced.ReducedPermutation method), 76
lengths() (sage.dynamics.interval_exchanges.iet.IntervalExchangeTransformation method), 113
letters() (sage.dynamics.interval_exchanges.template.Permutation method), 86
letters() (sage.dynamics.interval_exchanges.template.Permutation method), 86
letters() (sage.dynamics.interval_exchanges.template.RauzyDiagram method), 107
lift_to_rational_periodic() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 230
list() (sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutation method), 51
list() (sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutation method), 58
list() (sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutation method), 65
list() (sage.dynamics.interval_exchanges.labelled.LabelledPermutation method), 58
list() (sage.dynamics.interval_exchanges.labelled.LabelledPermutation method), 58
list() (sage.dynamics.interval_exchanges.labelled.LabelledPermutation method), 65
list() (sage.dynamics.interval_exchanges.labelled.LabelledPermutation method), 65
list() (sage.dynamics.interval_exchanges.labelled.LabelledPermutation method), 65
list() (sage.dynamics.interval_exchanges.reduced.FlippedReducedPermutationIET method), 73
list() (sage.dynamics.interval_exchanges.reduced.FlippedReducedPermutationIET method), 73
list() (sage.dynamics.interval_exchanges.reduced.FlippedReducedPermutationLI method), 73
list() (sage.dynamics.interval_exchanges.reduced.FlippedReducedPermutationLI method), 73
list() (sage.dynamics.interval_exchanges.reduced.ReducedPermutationIET method), 78
list() (sage.dynamics.interval_exchanges.reduced.ReducedPermutationLI method), 80
losers() (sage.dynamics.interval_exchanges.template.RauzyDiagram.Path method), 102
lr_inverse() (sage.dynamics.interval_exchanges.template.Permutation method), 87
Lxa() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 243
Lyb() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 244

M

mandelbrot_plot() (in module sage.dynamics.complex_dynamics.mandel_julia), 22
markov_chain() (sage.sandpiles.sandpile.Sandpile method), 132
matrix() (sage.dynamics.interval_exchanges.labelled.LabelledRauzyDiagram.Path method), 69
max_stable() (sage.sandpiles.sandpile.Sandpile method), 133
max_stable_div() (sage.sandpiles.sandpile.Sandpile method), 133
max_superstables() (sage.sandpiles.sandpile.Sandpile method), 134
min_cycles() (in module sage.sandpiles.sandpile), 166
min_recurrents() (sage.sandpiles.sandpile.Sandpile method), 134
minimal_model() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 205
multiplier() (sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine method), 182
multiplier() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 207
multiplier_spectra() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 208

N
nintervals() (sage.dynamics.flat_surfaces.strata.AbelianStratum method), 33
nintervals() (sage.dynamics.flat_surfaces.strata.ConnectedComponentOfAbelianStratum method), 35
NonHypCCA (in module sage.dynamics.flat_surfaces.strata), 39
NonHypConnectedComponentOfAbelianStratum (class in sage.dynamics.flat_surfaces.strata), 39
nonsink_vertices() (sage.sandpiles.sandpile.Sandpile method), 135
nonspecial_divisors() (sage.sandpiles.sandpile.Sandpile method), 135
normal_form() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 231
nth_iterate() (sage.dynamics.interval_exchanges.iet.IntervalExchangeTransformation method), 113
nth_iterate() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 209
nth_iterate_map() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 211
nth_iterate_psi() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeherK3Surface_ring method), 257

O
OddCCA (in module sage.dynamics.flat_surfaces.strata), 39
OddConnectedComponentOfAbelianStratum (class in sage.dynamics.flat_surfaces.strata), 39
orbit() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_affine method), 184
orbit() (sage.dynamics.arithmetic_dynamics.product_projective_ds.DynamicalSystem_product_projective method), 240
orbit() (sage.dynamics.arithmetic_dynamics.product_projective_ds.DynamicalSystem_projective method), 212
orbit_phi() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeherK3Surface_ring method), 256
orbit_psi() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeherK3Surface_ring method), 257
orbit_structure() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective Finite_field method), 186
orbit_structure() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective Finite_field method), 237
orbit_substitution() (sage.dynamics.interval_exchanges.labelled.LabelledRauzyDiagram.Path method), 69
order() (sage.sandpiles.sandpile.SandpileConfig method), 149
order_of_rauzy_action() (sage.dynamics.interval_exchanges.template.PermutationIET method), 97
out_degree() (sage.sandpiles.sandpile.Sandpile method), 135

P
parallel_firing_graph() (in module sage.sandpiles.sandpile), 166
parent() (sage.dynamics.flat_surfaces.strata.ConnectedComponentOfAbelianStratum method), 36
partition_sandpile() (in module sage.sandpiles.sandpile), 167
path() (sage.dynamics.interval_exchanges.template.RauzyDiagram method), 107
periodic_points() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 213
PeriodicSolitonCellularAutomata (class in sage.dynamics.cellular_automata.solitons), 1
Permutation (class in sage.dynamics.interval_exchanges.template), 84
Permutation() (in module sage.dynamics.interval_exchanges.constructors), 47
permutation() (sage.dynamics.interval_exchanges.iet.IntervalExchangeTransformation method), 113
PermutationIET (class in sage.dynamics.interval_exchanges.template), 91
PermutationLI (class in sage.dynamics.interval_exchanges.template), 99
Permutations_iterator() (in module sage.dynamics.interval_exchanges.constructors), 48
phi() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 257
picard_representatives() (sage.sandpiles.sandpile.Sandpile method), 136
plot() (sage.dynamics.interval_exchanges.iet.IntervalExchangeTransformation method), 114
plot_function() (sage.dynamics.interval_exchanges.iet.IntervalExchangeTransformation method), 114
plot_two_intervals() (sage.dynamics.interval_exchanges.iet.IntervalExchangeTransformation method), 114
points() (sage.sandpiles.sandpile.Sandpile method), 136
polytope() (sage.sandpiles.sandpile.SandpileDivisor method), 158
polytope_integer_pts() (sage.sandpiles.sandpile.SandpileDivisor method), 159
pop() (sage.dynamics.interval_exchanges.template.RauzyDiagram.Path method), 102
possible_periods() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 216
possible_periods() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_finite_field method), 238
postulation() (sage.sandpiles.sandpile.Sandpile method), 136
primes_of_bad_reduction() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 217
print_state() (sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata method), 12
print_state_evolution() (sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata method), 12
print_states() (sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata method), 13
psi() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 258
Q
q_reduced() (sage.sandpiles.sandpile.SandpileDivisor method), 159
QuadraticStratum (class in sage.dynamics.flat_surfaces.quadratic_strata), 40
Qxa() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 244
Qyb() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 244
R
Ramification_poly() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 245
random_DAG() (in module sage.sandpiles.sandpile), 167
random_WehlerK3Surface() (in module sage.dynamics.arithmetic_dynamics.wehlerK3), 261
range_singularities() (sage.dynamics.interval_exchanges.iet.IntervalExchangeTransformation method), 115
rank() (sage.sandpiles.sandpile.Sandpile.SandpileDivisor method), 159
rational_periodic_points() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 232
rational_preperiodic_graph() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 233
rational_preperiodic_points() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 234
rauzy_diagram() (sage.dynamics.flat_surfaces.strata.ConnectedComponentOfAbelianStratum method), 36
rauzy_diagram() (sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutationIET method), 53
rauzy_diagram() (sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutationLI method), 55
rauzy_diagram() (sage.dynamics.interval_exchanges.labelled.LabelledPermutationIET method), 61
rauzy_diagram() (sage.dynamics.interval_exchanges.labelled.LabelledPermutationLI method), 66
rauzy_diagram() (sage.dynamics.interval_exchanges.reduced.FlippedReducedPermutationIET method), 73
rauzy_diagram() (sage.dynamics.interval_exchanges.reduced.FlippedReducedPermutationLI method), 74
rauzy_diagram() (sage.dynamics.interval_exchanges.reduced.ReducedPermutationIET method), 78
rauzy_diagram() (sage.dynamics.interval_exchanges.reduced.ReducedPermutationLI method), 80
rauzy_move() (sage.dynamics.interval_exchanges.iet.IntervalExchangeTransformation method), 115
rauzy_move() (sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutationIET method), 53
rauzy_move() (sage.dynamics.interval_exchanges.labelled.LabelledPermutationIET method), 62
rauzy_move() (sage.dynamics.interval_exchanges.template.Permutation method), 87
rauzy_move_interval_substitution() (sage.dynamics.interval_exchanges.labelled.LabelledPermutationIET method), 62
rauzy_move_loser() (sage.dynamics.interval_exchanges.labelled.LabelledPermutation method), 59
rauzy_move_matrix() (sage.dynamics.interval_exchanges.labelled.LabelledPermutation method), 59
rauzy_move_relabel() (sage.dynamics.interval_exchanges.reduced.ReducedPermutationIET method), 78
rauzy_move_winner() (sage.dynamics.interval_exchanges.labelled.LabelledPermutation method), 60
RauzyDiagram (class in sage.dynamics.interval_exchanges.template), 100
RauzyDiagram() (in module sage.dynamics.interval_exchanges.constructors), 49
RauzyDiagram.Path (class in sage.dynamics.interval_exchanges.template), 101
recurrents() (sage.sandpiles.sandpile.Sandpile method), 136
reduced() (sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutationIET method), 54
reduced() (sage.dynamics.interval_exchanges.labelled.LabelledPermutationIET method), 55
reduced() (sage.dynamics.interval_exchanges.labelled.LabelledPermutationIET method), 63
reduced() (sage.dynamics.interval_exchanges.labelled.LabelledPermutationLI method), 66
reduced_form() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 218
reduced_laplacian() (sage.sandpiles.sandpile.Sandpile.Sandpile method), 137
ReducedPermutation (class in sage.dynamics.interval_exchanges.reduced), 74
ReducedPermutationIET (class in sage.dynamics.interval_exchanges.reduced), 79
ReducedPermutationsIET_iterator() (in module sage.dynamics.interval_exchanges.reduced), 80
ReducedRauzyDiagram (class in sage.dynamics.interval_exchanges.reduced), 80
reorder_vertices() (sage.sandpiles.sandpile.Sandpile.Sandpile method), 137
representative() (sage.dynamics.flat_surfaces.strata.ConnectedComponentOfAbelianStratum method), 36
representative() (sage.dynamics.flat_surfaces.strata.EvenConnectedComponentOfAbelianStratum method), 37
representative() (sage.dynamics.flat_surfaces.strata.HypConnectedComponentOfAbelianStratum method), 38
representative() (sage.dynamics.flat_surfaces.strata.OddConnectedComponentOfAbelianStratum method), 39
reset() (sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata method), 15
resolution() (sage.sandpiles.sandpile.Sandpile.Sandpile method), 138
resultant() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 222
right_composition() (sage.dynamics.interval_exchanges.template.RauzyDiagram.Path method), 103
right_rauzy_move() (sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutationIET method), 55
right_rauzy_move() (sage.dynamics.interval_exchanges.labelled.LabelledPermutationIET method), 66
right_rauzy_move() (sage.dynamics.interval_exchanges.reduced.FlippedReducedPermutation method), 72
right_rauzy_move() (sage.dynamics.interval_exchanges.reduced.ReducedPermutation method), 76
ring() (sage.sandpiles.sandpile.Sandpile.Sandpile method), 138

S
sage.dynamics.arithmetic_dynamics.affine_ds (module), 177
sage.dynamics.arithmetic_dynamics.generic_ds (module), 171
sage.dynamics.arithmetic_dynamics.product_projective_ds (module), 239
sage.dynamics.arithmetic_dynamics.projective_ds (module), 187
sage.dynamics.arithmetic_dynamics.wehlerK3 (module), 241
sage.dynamics.cellular_automata.solitons (module), 1
sage.dynamics.complex_dynamics.mandel_julia (module), 19
sage.dynamics.flat_surfaces.quadratic_strata (module), 40
sage.dynamics.flat_surfaces.strata (module), 25
Index

sage.dynamics.interval_exchanges.constructors (module), 41
sage.dynamics.interval_exchanges.iet (module), 109
sage.dynamics.interval_exchanges.labelled (module), 51
sage.dynamics.interval_exchanges.reduced (module), 72
sage.dynamics.interval_exchanges.template (module), 82
sage.sandpiles.sandpile (module), 117
sandlib() (in module sage.sandpiles.sandpile), 168
Sandpile (class in sage.sandpiles.sandpile), 122
sandpile() (sage.sandpiles.sandpile.SandpileConfig method), 149
sandpile() (sage.sandpiles.sandpile.SandpileDivisor method), 160
SandpileConfig (class in sage.sandpiles.sandpile), 143
SandpileDivisor (class in sage.sandpiles.sandpile), 151
separatrix_diagram() (sage.dynamics.interval_exchanges.template.PermutationIET method), 97
show() (sage.dynamics.interval_exchanges.iet.IntervalExchangeTransformation method), 115
show() (sage.sandpiles.sandpile.Sandpile method), 139
show() (sage.sandpiles.sandpile.SandpileConfig method), 150
show() (sage.sandpiles.sandpile.SandpileDivisor method), 160
show3d() (sage.sandpiles.sandpile.Sandpile method), 139
side_conversion() (in module sage.dynamics.interval_exchanges.template), 108
sigma_invariants() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 222
sigmaX() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeihlerK3Surface_ring method), 258
sigmaY() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeihlerK3Surface_ring method), 260
simulate_threshold() (sage.sandpiles.sandpile.SandpileDivisor method), 161
singularities() (sage.dynamics.interval_exchanges.iet.IntervalExchangeTransformation method), 115
sink() (sage.sandpiles.sandpile.Sandpile method), 139
smith_form() (sage.sandpiles.sandpile.Sandpile method), 139
SolitonCellularAutomata (class in sage.dynamics.cellular_automata.solitons), 4
solve() (sage.sandpiles.sandpile.Sandpile method), 140
specialization() (sage.dynamics.arithmetic_dynamics.generic_ds.DynamicalSystem method), 177
stabilize() (sage.sandpiles.sandpile.SandpileConfig method), 150
stabilize() (sage.sandpiles.sandpile.SandpileDivisor method), 161
stable_configs() (sage.sandpiles.sandpile.Sandpile method), 140
start() (sage.dynamics.interval_exchanges.template.RauzyDiagram.Path method), 103
state_evolution() (sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata method), 16
stationary_density() (sage.sandpiles.sandpile.Sandpile method), 141
str() (sage.dynamics.interval_exchanges.template.FlippedPermutation method), 82
str() (sage.dynamics.interval_exchanges.template.Permutation method), 88
stratum() (sage.dynamics.interval_exchanges.template.PermutationIET method), 97
substitution() (sage.dynamics.interval_exchanges.labelled.LabelledRauzyDiagram.Path method), 70
superstables() (sage.sandpiles.sandpile.Sandpile method), 141
support() (sage.sandpiles.sandpile.SandpileConfig method), 151
support() (sage.sandpiles.sandpile.SandpileDivisor method), 161
Sxa() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeihlerK3Surface_ring method), 245
Syb() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeihlerK3Surface_ring method), 246
symmetric() (sage.dynamics.interval_exchanges.template.Permutation method), 89
symmetric_recurrence() (sage.sandpiles.sandpile.Sandpile method), 142

T

tb_inverse() (sage.dynamics.interval_exchanges.template.Permutation method), 90
to_permutation() (sage.dynamics.interval_exchanges.template.PermutationIET method), 99
top_bottom_inverse() (sage.dynamics.interval_exchanges.template.Permutation method), 90
triangle_sandpile() (in module sage.sandpiles.sandpile), 168
tutte_polynomial() (sage.sandpiles.sandpile.Sandpile method), 142
twin_list_iet() (in module sage.dynamics.interval_exchanges.template), 109
twin_list_li() (in module sage.dynamics.interval_exchanges.template), 109

U
unsaturated_ideal() (sage.sandpiles.sandpile.Sandpile method), 142
unstable() (sage.sandpiles.sandpile.SandpileConfig method), 151
unstable() (sage.sandpiles.sandpile.SandpileDivisor method), 162

V
values() (sage.sandpiles.sandpile.SandpileConfig method), 151
values() (sage.sandpiles.sandpile.SandpileDivisor method), 162
version() (sage.sandpiles.sandpile.Sandpile static method), 143
vertex_iterator() (sage.dynamics.interval_exchanges.template.RauzyDiagram method), 107
vertical_inverse() (sage.dynamics.interval_exchanges.template.Permutation method), 90
vertices() (sage.dynamics.interval_exchanges.template.RauzyDiagram method), 108

W
WehlerK3Surface() (in module sage.dynamics.arithmetic_dynamics.wehlerK3), 241
WehlerK3Surface_field (class in sage.dynamics.arithmetic_dynamics.wehlerK3), 242
WehlerK3Surface_finite_field (class in sage.dynamics.arithmetic_dynamics.wehlerK3), 242
WehlerK3Surface_ring (class in sage.dynamics.arithmetic_dynamics.wehlerK3), 242
weierstrass_div() (sage.sandpiles.sandpile.SandpileDivisor method), 162
weierstrass_gap_seq() (sage.sandpiles.sandpile.SandpileDivisor method), 163
weierstrass_pts() (sage.sandpiles.sandpile.SandpileDivisor method), 163
weierstrass_rank_seq() (sage.sandpiles.sandpile.SandpileDivisor method), 164
weil_restriction() (sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine_field method), 185
wilmes_algorithm() (in module sage.sandpiles.sandpile), 168
winners() (sage.dynamics.interval_exchanges.template.RauzyDiagram.Path method), 103

Z
zero_config() (sage.sandpiles.sandpile.Sandpile method), 143
zero_div() (sage.sandpiles.sandpile.Sandpile method), 143