Discrete dynamics

Release 10.1

The Sage Development Team

Aug 21, 2023
<table>
<thead>
<tr>
<th></th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cellular Automata</td>
</tr>
<tr>
<td>2</td>
<td>Plotting of Mandelbrot and Julia Sets</td>
</tr>
<tr>
<td>3</td>
<td>Discrete dynamical systems</td>
</tr>
<tr>
<td>4</td>
<td>Sandpiles</td>
</tr>
<tr>
<td>5</td>
<td>Arithmetic Dynamical Systems</td>
</tr>
<tr>
<td>6</td>
<td>Indices and Tables</td>
</tr>
<tr>
<td></td>
<td>Python Module Index</td>
</tr>
<tr>
<td></td>
<td>Index</td>
</tr>
</tbody>
</table>
1.1 Catalog of Cellular Automata

The `cellular_automata` object may be used to access examples of various cellular automata currently implemented in Sage. Using tab-completion on this object is an easy way to discover and quickly create the cellular automata that are available (as listed here).

Let <tab> indicate pressing the Tab key. So begin by typing `cellular_automata.<tab>` to see the currently implemented named cellular automata.

- `cellular_automata.Elementary`
- `cellular_automata.GraftalLace`
- `cellular_automata.PeriodicSoliton`
- `cellular_automata.Soliton`

1.2 Elementary Cellular Automata

AUTHORS:
- Travis Scrimshaw (2018-07-07): Initial version

```python
class sage.dynamics.cellular_automata.elementary.ElementaryCellularAutomata

Bases: SageObject

Elementary cellular automata.

An elementary cellular automaton is a 1-dimensional cellular deterministic automaton with two possible values: $X := \{0, 1\}$. A state is therefore a sequence $s \in X^n$, and the evolution of a state $s \rightarrow s'$ is given for $s'_i$ by looking at the values at positions $s_{i-1}, s_i, s_{i+1}$ and is determined by the rule $0 \leq r \leq 255$ as follows. Consider the binary representation $r = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$. Then, we define $s'_i = b_j$, where $j = s_{i-1}s_is_{i+1}$ is the corresponding binary representation. In other words, the value $s'_i$ is given according to the following table:

<table>
<thead>
<tr>
<th>$b_7$</th>
<th>$b_6$</th>
<th>$b_5$</th>
<th>$b_4$</th>
<th>$b_3$</th>
<th>$b_2$</th>
<th>$b_1$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>110</td>
<td>101</td>
<td>100</td>
<td>011</td>
<td>010</td>
<td>001</td>
<td>000</td>
</tr>
</tbody>
</table>

We consider the boundary values of $s_0 = s_{n+1} = 0$.

INPUT:
• rule – an integer between 0 and 255
• width – (optional) the width of the ECA
• initial_state – (optional) the initial state given as a list of 0 and 1
• boundary – (default: (0, 0)) a tuple of the left and right boundary conditions respectively or None for
  periodic boundary conditions

Either width or initial_state must be given. If width is less than the length of initial_state, then
initial_state has 0 prepended so the resulting list has length width. If only width is given, then the initial
state is constructed randomly.

The boundary conditions can either be 0, 1, or a function that takes an integer n corresponding to the state and
outputs either 0 or 1.

EXAMPLES:
We construct an example with rule \( r = 90 \) using \( n = 20 \). The initial state consists of a single 1 in the rightmost entry:

```
sage: ECA = cellular_automata.Elementary(90, width=20, initial_state=[1])
sage: ECA.evolve(20)
sage: ascii_art(ECA)
```

We now construct it with different boundary conditions. The first is with the left boundary being 1 (instead of 0):

```
sage: ECA = cellular_automata.Elementary(90, width=20, initial_state=[1], boundary=(1,0))
sage: ECA.evolve(20)
sage: ascii_art(ECA)
```

(continues on next page)
Now we consider the right boundary as being 1 on every third value:

```python
sage: def rbdry(n):  return 1 if n % 3 == 0 else 0
sage: ECA = cellular_automata.Elementary(90, width=20, initial_state=[1],
  boundary=(0, rbdry))
```

Lastly we consider it with periodic boundary condition:

```python
sage: ECA = cellular_automata.Elementary(90, width=20, initial_state=[1],
  boundary=None)
```
sage: ascii_art(ECA)

```
X
X X
X X
X X X X
X X X X
X X X X X X X X
X X
X X X X
X X
X X X X
X X X X
X X X X X X X X
X X
X X X X
X X
X X X X
X X X X
X X X X X X X X
X X
X X X X
X X
X X X X
X X X X
X X X X X X X X
X X
X X X X
X X
X X X X
X X X X
X X X X X X X X
X X
X X X X
X X
X X X X
X X X X
X X X X X X X X
```

We show the local evolution rules for rule 110:

```
sage: for t in cartesian_product([[0,1],[0,1],[0,1]]):
    ....:     ECA = cellular_automata.Elementary(110, list(t))
    ....:     ECA.print_states(2)
    ....:     print('#')
```

```
#
X
XX
#
X
XX
#
XX
XXX
#
X
X
#
X X
XXX
#
XX
XX
#
XXX
X X
```
We construct an elementary cellular automaton with a random initial state with $n = 15$ and see the state after 50 evolutions:

```sage
ECA = cellular_automata.Elementary(26, width=25)
ECA.print_state(50)  # random
```

We construct and plot a larger example with rule 60:

```sage
ECA = cellular_automata.Elementary(60, width=200)
ECA.evolve(200)
ECA.plot()
```

With periodic boundary condition for rule 90:

```sage
ECA = cellular_automata.Elementary(90, initial_state=[1]+[0]*254+[1],
  boundary=None)
ECA.evolve(256)
ECA.plot()
```

1.2. Elementary Cellular Automata
REFERENCES:

Wikipedia article Elementary_cellular_automaton

evolve(number=None)

Evolve self.

INPUT:

• number – (optional) the number of times to perform the evolution

EXAMPLES:

```sage
ECA = cellular_automata.Elementary(110, [1,0,0,1,1,0,0,1,0,1])
sage: ascii_art(ECA)
X XX X X
sage: ECA.evolve()
sage: ascii_art(ECA)
X XX X X
X XXX XXXX
sage: ECA.evolve(10)
sage: ascii_art(ECA)
X XX X X
X XXX XXXX
XXX XXX X
X XXX X XX
XXX XXXXXX
X XXX X
XXX X XX
X XXX XXX
XXX X XX X
X XXXXXXX
XXX X
X X XX
```

plot(number=None)

Return a plot of self.

INPUT:

• number – the number of states to plot

EXAMPLES:

```sage
ECA = cellular_automata.Elementary(110, width=256)
sage: ECA.evolve(256)
sage: ECA.plot()
Graphics object consisting of 1 graphics primitive
```

print_state(number=None)

Print the state number.

INPUT:

• number – (default: the current state) the state to print

EXAMPLES:
print_states(number=None)
Print the first number states of self.

Note: If the number of states computed for self is less than number, then this evolves the system using the default time evolution.

INPUT:
• number – the number of states to print

EXAMPLES:

```python
sage: ECA = cellular_automata.Elementary(110, width=10,
.....: initial_state=[1,0,0,1,1,0,1])
```
```python
sage: ECA.print_state(15)
X X XXXXX
```
```python
sage: ECA.print_state(10)
X X XX
```
```python
sage: ECA.print_state(20)
X XXX
```
```python
sage: for i in range(11):
.....: ECA.print_state(i)
```
```python
X XX X
XX XXXXX
XXXXX X
XX X XX
XX XX XXX
XX XXXXX X
XXXX XXX
X X XX X
X X XXXXX
XXXXXXX X
X X XX
```

```python
print_states(number=None)
```
1.3 Graftal Lace Cellular Automata

AUTHORS:

• Travis Scrimshaw (2020-04-30): Initial version

class sage.dynamics.cellular_automata.glca.GraftalLaceCellularAutomata(rule)
Bases: SageObject

Graftal Lace Cellular Automata (GLCA).

A GLCA is a deterministic cellular automaton whose rule is given by an 8-digit octal number \( r_7 \cdots r_0 \). For a node \( s_i \), let \( b_k \), for \( k = -1, 0, 1 \) denote if there is an edge from \( s_i \) to \( s'_i+k \), where \( s'_i \) is the previous row. We determine the value at \( t_{i+k} \) by considering the value of \( r_m \), where the binary representation of \( m \) is \( b'_{-1} b_0 b'_{1} \). If \( r_m \) has a binary representation of \( b'_{-1} b_0 b'_{1} \), then we add \( b'_k \) to \( t_{i+k} \).

INPUT:

• rule – a list of length 8 with integer entries \( 0 \leq x < 8 \)

EXAMPLES:

```
sage: G = cellular_automata.GraftalLace([0,2,5,4,7,2,3,3])
sage: G.evolve(3)
sage: ascii_art(G)
  o |
  | o
  | o o o
  /| /|
  o o o o o
  /| /|
  o o o o o
  o o o o o

sage: G = cellular_automata.GraftalLace([3,0,3,4,7,6,3,1])
sage: G.evolve(3)
sage: ascii_art(G)
  o |
  | o  \|
  | o o o
  /| /|/\|/|
  o o o o o
  /| /|/\|/|
  o o o o o

sage: G = cellular_automata.GraftalLace([2,0,3,3,6,0,2,7])
sage: G.evolve(20)
sage: G.plot()
Graphics object consisting of 842 graphics primitives
```

REFERENCES:

• [Kas2018]
evolve(number=None)
Evolve self.

INPUT:
• number – (default: 1) the number of times to perform the evolution

EXAMPLES:

```
sage: G = cellular_automata.GraftalLace([5,1,2,5,4,5,5,0])
sage: ascii_art(G)
     o
     |
     o
sage: G.evolve(2)
sage: ascii_art(G)
     o     \    /    
     |     o  o    
     o   /  /  /  
    / \  / \  / \ 
   /   /   /   / 
  o   o   o   o   o
sage: G = cellular_automata.GraftalLace([0,2,1,4,7,2,3,0])
sage: G.evolve(3)
sage: ascii_art(G)
     o     \    /    
     |     o  o    
     o   /  /  /  
    / \  / \  / \ 
   /   /   /   / 
  o   o   o   o   o
```

plot(number=None)
Return a plot of self.

INPUT:
• number – the number of states to plot

EXAMPLES:

```
sage: G = cellular_automata.GraftalLace([5,1,2,5,4,5,5,0])
sage: G.evolve(20)
sage: G.plot()
Graphics object consisting of 865 graphics primitives
```

print_states(number=None, use_unicode=False)
Print the first num states of self.

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

INPUT:

• number – the number of states to print

EXAMPLES:

```
sage: G = cellular_automata.GraftalLace([5, 1, 2, 5, 4, 5, 5, 0])
sage: G.evolve(2)
sage: G.print_states()
  0
  | 0
 / \ o o o
 / \ / \ o o o o
sage: G.evolve(20)
sage: G.print_states(3)
  0
  | 0
 / \ o o o
 / \ / \ o o o o
```

1.4 Soliton Cellular Automata

AUTHORS:

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

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

Bases: 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 $p = p_t \otimes \cdots \otimes p_1 \otimes p_0$ and the *time evolution* $T_s$ is defined by

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

for some element $u \in B^{-\sigma}$.

INPUT:

• initial_state – the list of elements, can also be a string when vacuum is 1 and $n$ is $\mathfrak{sl}_n$
• cartan_type – (default: 2) the value $n$, for $\mathfrak{sl}_n$, 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`:

```
sage: P = PeriodicSolitonCellularAutomata('"1123334112411142311141112311", 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
evolutuions: [(1, 31)]
current state:
34 ......24....243....4.223.233.
```
sage: P.evolve(carrier_capacity=2)
sage: P.evolve(carrier_index=2)
sage: P.evolve(carrier_index=2, carrier_capacity=3)
sage: P.print_states(10)
```
t: 0
..23334...24....423..4...23..2
t: 1
34 ......24....243....4.223.233.
t: 2
......24....24.3....4223.2333.4
t: 3
......34....34.2..223234.24..3.
t: 4
......34..23..242223.4..33....4.
t: 5
....34.2223.224.3....4.33.....4..
t: 6
34223....24....3....433......4.22
t: 7
23....24....3....343....22234..
t: 8
....24.....3..34.322244....3..23
t: 9
..24.....332442342........3.23..
```

Using \( r = 2 \) in type \( A_3^{(1)} \):

```
sage: initial = [[2,1],[2,1],[4,1],[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
```

(continues on next page)
We do some examples in other types:

```python
sage: initial = [[1], [2], [1], [1], [1], [3], [1], ['E'], [1], [1]]
sage: P = PeriodicSolitonCellularAutomata(initial, ['D', 4, 3])
sage: P.print_states(10)
t: 0
   .22...3.E..
t: 1
   2....3.E..2
t: 2
   ....3.E.22.
t: 3
   ...3.E22...
t: 4
   ....32E2...
t: 5
   .00.2......
t: 6
   _22.2......
t: 7
   2.2......3E
t: 8
   .2......30.2
t: 9
   2....332.2.
```

```python
sage: P = PeriodicSolitonCellularAutomata([[3], [2], [1], [1], [-2]], ['C', 2, 1])
sage: P.print_state_evolution(0)

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 \(B^{r,s}\).

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.

**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:**

```
sage: P = PeriodicSolitonCellularAutomata('12411133214131221122', 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)
..33.22.42322...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
 ..33.22.42322...22...4
t: 3
 .33.22.42322...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.....43322
```

```
t: 8
 ..4322.2.....433222.
```

(continues on next page)
Discrete dynamics, Release 10.1

(continued from previous page)

t: 9
  .4322.2.....433222..

\[ t: 9 \]
\[
\begin{array}{c}
sage: P = PeriodicSolitonCellularAutomata('12411132121', 4) \\
sage: P.evolve(carrier_index=2, carrier_capacity=3) \\
sage: P.state_evolution(0)
\end{array}
\]
\[
\begin{bmatrix}
[[1, 1, 1], [2, 2, 4]], \\
[[1, 1, 2], [2, 2, 4]], \\
[[1, 1, 3], [2, 2, 4]], \\
[[1, 1, 1], [2, 2, 3]], \\
[[1, 1, 1], [2, 2, 3]], \\
[[1, 1, 1], [2, 2, 3]], \\
[[1, 1, 2], [2, 2, 2]], \\
[[1, 1, 1], [2, 2, 2]], \\
[[1, 1, 1], [2, 2, 2]], \\
[[1, 1, 1], [2, 2, 4]], \\
[[1, 1, 1], [2, 2, 4]]
\end{bmatrix}
\]

\[
\text{class } \text{sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata}(\text{initial_state}, \\
\text{cartan_type}=2, \\
\text{vacuum}=1)
\]

Bases: SageObject

Soliton cellular automata.

Fix an affine Lie algebra \( \mathfrak{g} \) 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 \( R(p_1 \otimes u^{(i)}) = u^{(i+1)} \otimes p_i \) with \( u^{(0)} = u_s \) and \( T_s(p) = \cdots \otimes p_1 \otimes p_0 \). 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:

- \text{initial_state} – the list of elements, can also be a string when vacuum is 1 and \( n = sl_n \)
- \text{cartan_type} – (default: 2) the value \( n \), for \( sl_n \), or a Cartan type
- \text{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 \( A_3 \)):

\[
\text{sage: B = SolitonCellularAutomata('34111112224111112223', 4)} \\
\text{sage: B}
\]

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

\[
\begin{bmatrix}
\text{initial state:} \\
34......224....2223 \\
\text{evolutions:} [] \\
\text{current state:} \\
34......224....2223
\end{bmatrix}
\]
We then apply an standard evolution:

\begin{verbatim}
sage: B.evolve()
sage: B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
   initial state:
   34......224....2223
evolution: [(1, 19)]
current state:
....................34....224...2223....
\end{verbatim}

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

\begin{verbatim}
sage: B.evolve(3)
sage: B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
   initial state:
   34......224....2223
evolution: [(1, 19), (1, 3)]
current state:
....................34....224...2223....
\end{verbatim}

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:

\begin{verbatim}
sage: B.evolve(carrier_capacity=7, carrier_index=3)
Last carrier:
  1 1 1 1 1 1 1
  2 2 2 2 2 3 3
  3 3 3 3 3 4 4
sage: B
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....
\end{verbatim}

To summarize our current evolutions, we can use \texttt{print_states()}:

\begin{verbatim}
sage: B.print_states(5)
\end{verbatim}

(continues on next page)
To run the SCA further under the standard evolutions, one can use \texttt{print\_states()} or \texttt{latex\_states()}:

\begin{verbatim}
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...223...2222.......t: 7
          ..............................................2222...23...222..........t: 8
          ..............................................2222....23..222.........t: 9
          ..............................................2222....23..222.........t: 10
          ..............................................2222....23..222.........t: 11
          ..............................................2222....23..222.........t: 12
          ..............................................2222....23..222.........t: 13
          ..............................................2222....23..222.........t: 14
          ..............................................2222....23..222.........
\end{verbatim}

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:

\begin{verbatim}
sage: B = SolitonCellularAutomata([[4,1],[4,1],[2,1],[2,1],[2,1],[2,1],[3,1],[3,1],...
˓→[3,2]], 4, 2)
\end{verbatim}

We perform 3 evolutions and obtain the following:
We construct Example 2.9 from [LS2017]:

```
sage: B = SolitonCellularAutomata([[2],[-3],[1],[1],[1],[4],[0],[-2],
.....: [1],[1],[1],[1],[3],[-4],[-3],[-3],[1]], ['D',5,2])
sage: B.print_states(10)
t: 0
  ........................................23...402..3433.
t: 1
  ........................................23..402..3433.....
t: 2
  ........................................23.402..3433....
t: 3
  ........................................243.02.3433.....
t: 4
  ........................................2403.02.3433..
t: 5
  ........................................2403..42333..
t: 6
  ........................................2403..42433..
t: 7
  ........................................2403..44243.
t: 8
  ........................................2403..44243.
t: 9
  ........................................2403....44243.
```

Example 3.4 from [LS2017]:

```
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)
t: 0
  ........................................E...30....231.
t: 1
  ........................................E..30...231..
t: 2
  ........................................E..30...231..
t: 3
  ........................................E303.21..
t: 4
  ........................................303E2.22..
t: 5
  ........................................303E...222.
```

(continues on next page)
Example 3.12 from [LS2017]:

```python
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)
```

```
-1 -3-2
3 2-3
2 . . 1 1
-1-3 2
3 2-3
2 1 1 ...
-3-1
```

```
t: 0 3 2-3
```

```
t: 1 3 2-3
```
**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 $B^{r \times s}$.

**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:**

```python
sage: B = SolitonCellularAutomata('341111122411112223', 4)
sage: for k in range(10):
    ....:     B.evolve()
    ....: B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
initial state:
34......224....2223
evolutions: [(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
evolutions: [(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....2223
evolutions: [(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])
```

(continues on next page)
sage: B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
initial state:
34........224....2223
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....2223
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....2223
evolutuions: [(2, 1), (2, 1), (2, 1)]
current state:
..24......222.....2222.

latex_state_evolution(num, scale=1)

Return a latex version of the evolution process of the state num.

See also:

state_evolution(), print_state_evolution()

EXAMPLES:
Discrete dynamics, Release 10.1

```python
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**(num=None, as_array=True, box_width='5pt')

Return a latex version of the states.

**INPUT:**

- **num** – the number of states
- **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
- **box_width** – (default: '5pt') the width of the . used to represent the vacuum state when as_array is True

If as_array is False, then the vacuum element is printed in a gray color. If as_array is True, then the vacuum is given as a.

Use the box_width to help create more even spacing when a column in the output contains only vacuum elements.

**EXAMPLES:**

```python
sage: B = SolitonCellularAutomata('411122', 4)
sage: B.latex_states(8)
{\arraycolsep=0.5pt \begin{array}{c|ccccccccccccccccccc}
t = 0 & \cdots & ... & \makebox[5pt]{} & 4 & \makebox[5pt]{} & \makebox[5pt]{} & \makebox[5pt]{} & 2 & 2 \\
t = 1 & \cdots & ... & 4 & \makebox[5pt]{} & \makebox[5pt]{} & \makebox[5pt]{} & \makebox[5pt]{} & 2 & 2 & ... \\
t = 2 & \cdots & ... & 4 & \makebox[5pt]{} & \makebox[5pt]{} & \makebox[5pt]{} & \makebox[5pt]{} & 2 & 2 & ... \\
t = 3 & \cdots & ... & 4 & 2 & 2 & ... \\
t = 4 & \cdots & ... & 2 & 4 & 2 & ... \\
t = 5 & \cdots & ... & 2 & 4 & \makebox[5pt]{} & 2 & ... \\
t = 6 & \cdots & ... & 2 & 4 & \makebox[5pt]{} & \makebox[5pt]{} & \makebox[5pt]{} \\
\end{array}}
sage: B = SolitonCellularAutomata('511122', 5)
sage: B.latex_states(8, as_array=False)
{\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|
```

(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('341111122411112223', 4)
sage: B.print_state()
34......224....2223
sage: B.evolve(number=2)
sage: B.print_state(vacuum_letter='_,

111 --+-- 111 --+-- 111 --+-- 113 --+-- 112 --+-- 123 --+-- 113 --+-- 111

sage: B.print_state(10, '_, True)
____2344_______222____23

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)
1 1 1 3 1 2 3
| | | | | | | | | | 111 ---- 111 --+-- 111 --+-- 113 --+-- 112 --+-- 123 --+-- 113 --+-- 111
| | | | | | | | | | 1 1 3 2 3 1 1
sage: B.print_state_evolution(1)
print_states(num=None, vacuum_letter=\')

Print the first num states of self.

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

INPUT:

* num – the number of states to print

EXAMPLES:

```
sage: B = SolitonCellularAutomata([[2],[-1],[1],[1],[1],[1],[2],[2],[3],...:[-2],[1],[1],[2],[-1],[1],[1],[1],[1],[1],[1],[2],[3],[3],[-3],[-2]],...:['C',3,1])
sage: B.print_states(7)
t: 0 ..........................21...232...21......23332
   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
t: 1 ..........................21...232...21......23332.....
   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
t: 2 ..........................21...232...21......23332.....
   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
t: 3 ..........................21...232...21......23332.....
   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
t: 4 ..........................21...232...21......23332.....
   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
t: 5 ..........................21...232...21......23332.....
   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
t: 6 ..........................21...232...21......23332.....
   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

```
sage: B = SolitonCellularAutomata([[2],[1],[1],[1],[3],[-2],[1],[1],...:[2],[2],[2],[-3],[1],[1],[1],[1],[1],[1],[2],[3],[3],[-3],...:['B',3,1])
sage: B.print_states(9, ' ')
t: 0 ..........................21...232...21......23332
   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
t: 1 ..........................21...232...21......23332.....
   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
t: 2 ..........................21...232...21......23332.....
   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
t: 3 ..........................21...232...21......23332.....
   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
t: 4 ..........................21...232...21......23332.....
   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
t: 5 ..........................21...232...21......23332.....
   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```


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)

(continues on next page)
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
   .........................03.........................
t: 1
   .........................3E.........................
t: 2
   .........................21.........................
t: 3
   .........................20E.........................
t: 4
   .........................233.........................
t: 5
   .........................2302.......................
t: 6
   .........................23322.....................
t: 7
   .........................2322......................
t: 8
   .........................233......................
t: 9
   .........................233......................
t: 10
   .........................2302.....................
t: 11
   .........................23322...................
t: 12
   .........................23322...................
t: 13
   .........................23322...................
t: 14
   .........................23322...................
```

reset()
Reset self back to the initial state.

EXAMPLES:

```python
sage: B = SolitonCellularAutomata('3411111224', 4)
sage: B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
    initial state:
    34......224
evolutuions: []
current state:
    34......224
sage: B.evolve()
sage: B.evolve()
sage: B.evolve()
sage: B.evolve()
sage: B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
    initial state:
    34......224
evolutuions: [(1, 11), (1, 11), (1, 11), (1, 11)]
current state:
    ...34..224............
sage: B.reset()
sage: B
Soliton cellular automata of type ['A', 3, 1] and vacuum = 1
    initial state:
    34......224
evolutuions: []
current state:
    34......224
```

**state_evolution**(num)

Return a list of the carrier values at state num evolving to the next state.

If 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:

- print_state_evolution()
- 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)
```

(continues on next page)
\[
\begin{array}{cccccccc}
1, & 1, & 1, & 1, & 1, & 1, & 1, & 1 \\
1, & 1, & 1, & 1, & 1, & 1, & 1, & 1 \\
1, & 1, & 1, & 1, & 1, & 1, & 1, & 1 \\
1, & 1, & 1, & 1, & 1, & 1, & 1, & 1 \\
1, & 1, & 1, & 1, & 1, & 1, & 1, & 1 \\
1, & 1, & 1, & 1, & 1, & 1, & 1, & 1 \\
1, & 1, & 1, & 1, & 1, & 1, & 1, & 3 \\
1, & 1, & 1, & 1, & 1, & 1, & 3, & 2 \\
1, & 1, & 1, & 1, & 1, & 1, & 1, & 1 \\
1, & 1, & 1, & 1, & 1, & 1, & 1, & 1 \\
1, & 1, & 1, & 1, & 1, & 1, & 1, & 1 \\
1, & 1, & 1, & 1, & 1, & 1, & 1, & 1 \\
1, & 1, & 1, & 1, & 1, & 1, & 1, & 1 \\
\end{array}
\]
2.1 Mandelbrot and Julia sets

Plots the Mandelbrot and Julia sets for general polynomial maps in the complex plane.

The Mandelbrot set is the set of complex numbers \( c \) for which the map \( f_c(z) \) 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 often an example of a fractal when plotted in the complex plane. For general one parameter families of polynomials, the mandelbrot set is the parameter values for which the orbits of all critical points remains bounded.

The Julia set for a given parameter \( c \) is the set of complex numbers for which the function \( f_c(z) \) 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:

- \( \text{theta} \) – double or list of doubles, angles between 0 and 1 inclusive.

kwds:

- \( \text{image} \) – 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 \( S*D \)). 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.
- \( \text{prec} \) – 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.
- \( \text{ray\_color} \) – 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:

```
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:

```
sage: M = mandelbrot_plot(x_center=0)  # not tested
sage: external_ray(5/7, image=M)  # not tested
```

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

```
sage: M = mandelbrot_plot(x_center=0)
sage: external_ray(5/7, x_center=0, image=M)
```

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

```
sage.dynamics.complex_dynamics.mandel_julia.julia_plot(f=None, **kwds)
```

Plots the Julia set of a given polynomial f. Users can specify whether they would like to display the Mandelbrot side by side with the Julia set with the mandelbrot argument. If f is not specified, this method defaults to \( f(z) = z^2 - 1 \).

The Julia set of a polynomial \( f \) is the set of complex numbers \( z \) for which the function \( f(z) \) 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:
Let \( R_c = (1 + \sqrt{1 + 4|c|})/2 \) if the polynomial is of the form \( f(z) = z^2 + c \); otherwise, let \( R_c = 2 \). For every \( p \in \mathbb{C} \), if \( |f^k(p)| > R_c \) for some \( k \geq 0 \), then \( f^n(p) \to \infty \). Let \( N \) be the maximum number of iterations. Compute the first \( N \) points on the orbit of \( p \) under \( f \). If for any \( k < N \), \( |f^k(p)| > R_c \), we stop the iteration and assign a color to the point \( p \) based on how quickly \( p \) escaped to infinity under iteration of \( f \). If \( |f^i(p)| \leq R_c \) for all \( i \leq N \), we assume \( p \) is in the Julia set and assign the point \( p \) the color black.

INPUT:

- \( f \) – input polynomial (optional - default: \( z^2 - 1 \)).
• **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 $f(z)$.

• **pixel_count** – long (optional - default: `500`), side length of image in number of pixels.

• **base_color** – hex color (optional - default: `'steelblue'`), color used to determine the coloring of set (any valid input for Color).

• **level_sep** – 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.

**Todo:** Implement the side-by-side Mandelbrot-Julia plots for general one-parameter families of polynomials.

**EXAMPLES:**

The default $f$ is $z^2 - 1$:

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

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

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

```
sage: R.<z> = CC[]  
sage: f = z^3 - z + 1  
sage: julia_plot(f)  # long time
500x500px 24-bit RGB image
```

To display an interactive plot of the Julia set in the Notebook, set `interact` to True. (This is only implemented for polynomials of the form $f = z^2 + c$):

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

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

```python
sage: julia_plot(period=[2,3])
```

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

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

Polynomial maps can be defined over a polynomial ring or a fraction field, so long as \( f \) is polynomial:

```python
sage: R.<z> = CC[]
sage: f = z^2 - 1
sage: julia_plot(f)
```

```python
sage: R.<z> = CC[]
sage: K = R.fraction_field(); z = K.gen()
sage: f = z^2-1
sage: julia_plot(f)
```

Interact functionality is not implemented if the polynomial is not of the form \( f = z^2 + c \):

```python
sage: R.<z> = CC[]
sage: f = z^3 + 1
sage: julia_plot(f, interact=True)
Traceback (most recent call last):
... NotImplementedError: The interactive plot is only implemented for ...
```

```python
sage.dynamics.complex_dynamics.mandel_julia.kneading_sequence(theta)
```

Determines the kneading sequence for an angle \( \theta \) in \( \text{RR}/\text{ZZ} \) which is periodic under doubling. We use the definition for the kneading sequence given in Definition 3.2 of [LS1994].

**INPUT:**

- \( \theta \) – a rational number with odd denominator

**OUTPUT:**

a string representing the kneading sequence of \( \theta \) in \( \text{RR}/\text{ZZ} \)

**REFERENCES:**
EXAMPLES:

```
sage: kneading_sequence(0)
'1**'
```

```
sage: kneading_sequence(1/3)
'1**'
```

Since 1/3 and 7/3 are the same in RR/ZZ, they have the same kneading sequence:

```
sage: kneading_sequence(7/3)
'1**'
```

We can also use (finite) decimal inputs, as long as the denominator in reduced form is odd:

```
sage: kneading_sequence(1.2)
'110**'
```

Since rationals with even denominator are not periodic under doubling, we have not implemented kneading sequences for such rationals:

```
sage: kneading_sequence(1/4)
Traceback (most recent call last):
  ...
ValueError: input must be a rational number with odd denominator
```

```python
sage.dynamics.complex_dynamics.mandel_julia.mandelbrot_plot(f=None, **kwds)
```

Plot of the Mandelbrot set for a one parameter family of polynomial maps.

The family $f_c(z)$ must have parent $R$ of the form $R.<z,c> = \mathbb{C}[]$.

REFERENCE:

[Dev2005]

INPUT:

- $f$ – map (optional - default: $z^2 + c$), polynomial family used to plot the Mandelbrot set.
- parameter – variable (optional - default: $c$), parameter variable used to plot the Mandelbrot set.
- 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 $f_c(z)$.
- pixel_count – long (optional - default: $500$), side length of image in number of pixels.
- base_color – RGB color (optional - default: $[40, 40, 40]$) color used to determine the coloring of set.
- level_sep – 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
```

```
sage: mandelbrot_plot(pixel_count=1000)
1000x1000px 24-bit RGB image
```

```
sage: mandelbrot_plot(x_center=-1.11, y_center=0.2283, image_width=1/128, max_iteration=2000, number_of_colors=500, base_color=[40, 100, 100])
500x500px 24-bit RGB image
```

To display an interactive plot of the Mandelbrot in the Notebook, set `interact` to `True`. (This is only implemented for \( z^2 + c \))

```
sage: mandelbrot_plot(interact=True)
...interactive(children=(FloatSlider(value=0.0, description='Real center', max=1.0, min=-1.0, step=1e-05), FloatSlider(value=0.0, description='Imag center', max=1.0, min=-1.0, step=1e-05), FloatSlider(value=4.0, description='Width', max=4.0, min=1e-05, step=1e-05), IntSlider(value=500, description='Iterations', max=1000), IntSlider(value=500, description='Pixels', max=1000, min=10), IntSlider(value=1, description='Color sep', max=20, min=1), IntSlider(value=30, description='# Colors', min=1), ColorPicker(value='#ff6347', description='Base color'), Output()), _dom_classes=('widget-interact',))
```

```
sage: mandelbrot_plot(interact=True, x_center=-0.75, y_center=0.25, image_width=1/2, number_of_colors=75)
...interactive(children=(FloatSlider(value=0.75, description='Real center', max=1.0, min=-1.0, step=1e-05), FloatSlider(value=0.25, description='Imag center', max=1.0, min=-1.0, step=1e-05), FloatSlider(value=0.5, description='Width', max=4.0, min=1e-05, step=1e-05), IntSlider(value=500, description='Iterations', max=1000), IntSlider(value=500, description='Pixels', max=1000, min=10), IntSlider(value=1, description='Color sep', max=20, min=1), IntSlider(value=75, description='# Colors', min=1), ColorPicker(value='#ff6347', description='Base color'), Output()), _dom_classes=('widget-interact',))
```

Polynomial maps can be defined over a multivariate polynomial ring or a univariate polynomial ring tower:

```
sage: R.<z,c> = CC[]
sage: f = z^2 + c
sage: mandelbrot_plot(f)
500x500px 24-bit RGB image
```

```
sage: B.<c> = CC[]
sage: R.<z> = B[
```
When the polynomial is defined over a multivariate polynomial ring it is necessary to specify the parameter variable (default parameter is $c$):

```
sage: R.<a,b> = CC[]
sage: f = a^2 + b^3
sage: mandelbrot_plot(f, parameter=b)  # long time
500x500px 24-bit RGB image
```

Interact functionality is not implemented for general polynomial maps:

```
sage: R.<z,c> = CC[]
sage: f = z^3 + c
sage: mandelbrot_plot(f, interact=True)
Traceback (most recent call last):
...
NotImplementedError: interact only implemented for $z^2 + c$
```
A discrete dynamical system (henceforth DDS) is a pair \((X, \phi)\) of a set \(X\) and a map \(\phi : X \to X\). (This is one of several things known as a “discrete dynamical system” in mathematics.)

This file implements the following classes for discrete dynamical systems:

- **DiscreteDynamicalSystem**: general discrete dynamical system, as above. Inherit from this class if the ground set of your DDS is infinite or large enough that you want to avoid it getting stored as a list. See the doc of this class for further details.

- **FiniteDynamicalSystem**: finite discrete dynamical system. This can be instantiated by calling `DiscreteDynamicalSystem` with the parameter `is_finite` set to `True`.

- **InvertibleDiscreteDynamicalSystem**: invertible discrete dynamical system. This implements an `inverse_evolution` method for \(\phi^{-1}\) (the default implementation simply applies \(\phi\) over and over until the original value is revisited; the last value before that is then taken to be the result). This can be instantiated by calling `DiscreteDynamicalSystem` with the parameter `inverse` provided.

- **InvertibleFiniteDynamicalSystem**: invertible finite discrete dynamical system. This can be instantiated by calling `DiscreteDynamicalSystem` with the parameter `is_finite` set to `True` and the parameter `inverse` provided.

**Todo:**

- Implement some more functionality for homomesy and invariance testing: Checking invariance on a sublist; computing the first \(k\) entries of an orbit (useful when orbits can be too large); `orbits_iterator` (for when there are too many orbits to list); etc.

- Further examples for non-auto functionality: e.g., infection on a chessboard; Conway’s game of life.

- Subclasses for DDSes whose ground set is an enumerated set. Should we have those?

- Implement caching for orbits (can be useful: some DDSes have a complicated evolution that shouldn’t be recomputed every time). Does this require a whole new class?

- Further functionality for non-invertible DDSes: `is_recurrent`, `recurrent_entries`, `idempotent_power`, etc.

- Wrap (some of) the `cyclic_sieving_phenomenon.py` methods (`sage.combinat.cyclic_sieving_phenomenon`).

- Interact with `sage.dynamics`. This requires someone who knows the latter part of the Sage library well.

```python
class sage.dynamics.finite_dynamical_system.DiscreteDynamicalSystem(X, phi, cache_orbits=False, create_tuple=False)
```
A discrete dynamical system (henceforth DDS) is a pair \((X, \phi)\) of a set \(X\) and a map \(\phi : X \to X\). This set \(X\) is called the ground set of the DDS, while the map \(\phi\) is called the evolution of the DDS.

A discrete dynamical system (short: DDS) is a pair \((X, \phi)\) of a set \(X\) and a map \(\phi : X \to X\). (This is one of several things known as a “discrete dynamical system” in mathematics.) Thus, a DDS is the same as an endomorphism of a set. The DDS is said to be finite if \(X\) is finite. The DDS is said to be invertible if the map \(\phi\) is invertible. The set \(X\) is called the ground set of the DDS; the map \(\phi\) is called the evolution of the DDS; the inverse map \(\phi^{-1}\) (when it exists) is called the inverse evolution of the DDS.

Given a DDS \((X, \phi)\), we can study

- its orbits (i.e., the lists \((s, \phi(s), \phi^2(s), \phi^3(s), \ldots)\) for \(s \in X\)),
- its invariants (i.e., maps \(f : X \to Y\) satisfying \(f \circ \phi = f\)),
- its cycles (i.e., lists \((u_1, u_2, \ldots, u_k)\) of elements of \(X\) such that \(\phi(u_i) = u_{i+1}\) for each \(i \leq k\), where we set \(u_{k+1} = u_1\)),
- its homomesies (i.e., maps \(h : X \to A\) to a \(\mathbb{Q}\)-vector space \(A\) such that the average of the values of \(h\) on each cycle is the same),

and various other features. (Some of these require \(X\) to be finite or at least to have finite orbits.)

**INPUT:**

- \(X\) – set, list, tuple, or another iterable, or \(\text{None}\) (default: \(\text{None}\)); the ground set for the DDS. This can be \(\text{None}\) (in which case Sage will not know the ground set, but can still apply evolution to any elements that are provided to it). Make sure to set the \text{create_tuple} argument to \text{True} if the \(X\) you provide is an iterator or a list, as otherwise your \(X\) would be exposed (and thus subject to mutation or exhaustion).
- \(\phi\) – function, or callable that acts like a function; the evolution of the DDS.
- \text{cache_orbits} – boolean (default: \text{False}); whether or not the orbits should be cached once they are computed. This currently does nothing, as we are not caching orbits yet.
- \text{create_tuple} – boolean (default: \text{False}); whether or not the input \(X\) should be translated into a tuple. Set this to \text{True} to prevent mutation if \(X\) is a list, and to prevent exhaustion if \(X\) is an iterator.
- \text{inverse} – function, or callable that acts like a function, or boolean or \(\text{None}\) (default: \(\text{None}\)); the inverse evolution of the DDS, if the DDS is invertible. Set this to \(\text{None}\) or \text{False} if the DDS is not invertible (or you don’t want Sage to treat it as such). Alternatively, by setting this argument to \text{True}, you can signal that the DDS is invertible without providing the inverse evolution. (In this case, Sage will compute the inverse, assuming the orbits to be finite.)
- \text{is_finite} – boolean or \(\text{None}\) (default: \(\text{None}\)); whether the DDS is finite. The default option \(\text{None}\) leaves this to Sage to decide. Only set this to \text{True} if you provide the ground set \(X\).

**EXAMPLES:**

The following discrete dynamical system is neither finite nor invertible:

```
sage: D = DiscreteDynamicalSystem(NN, lambda x: x + 2)
sage: D.ground_set() 
Non negative integer semiring 
sage: D.evolution()(5) 
7
sage: D.evolution_power(7)(5)
```
The necessity of `create_tuple=True`:

```python
sage: X = [0, 1, 2, 3, 4]
sage: D_wrong = DiscreteDynamicalSystem(X, lambda x: (x**3) % 5)
sage: D_right = DiscreteDynamicalSystem(X, lambda x: (x**3) % 5, create_tuple=True)
sage: X[4] = 666  # evil
sage: D_wrong.ground_set()
[0, 1, 2, 3, 666]
sage: D_right.ground_set()
(0, 1, 2, 3, 4)
```

Here is an invertible (but infinite) discrete dynamical system whose orbits are finite:

```python
sage: D = DiscreteDynamicalSystem(NN, lambda x: (x + 2 if x % 6 < 4 else x - 4), inverse=True)
sage: D.ground_set()
Non negative integer semiring
sage: D.evolution()(5)
1
sage: D.evolution()(1)
3
sage: D.evolution()(3)
5
sage: D.evolution_power(2)(5)
3
sage: D.evolution_power(3)(5)
5
sage: D.evolution_power(-2)(5)
1
sage: D.inverse_evolution()(4)
2
sage: D.orbit(3)
[3, 5, 1]
```

Setting the `inverse` parameter to `None` or `False` would give the same system without the functionality that relies on invertibility:

```python
sage: D = DiscreteDynamicalSystem(NN, lambda x: (x + 2 if x % 6 < 4 else x - 4), inverse=False)
sage: D.ground_set()
Non negative integer semiring
sage: D.evolution()(5)
1
sage: D.inverse_evolution()(4)
Traceback (most recent call last):
  ...
AttributeError: 'DiscreteDynamicalSystem' object has no attribute 'inverse_evolution'
```

(continues on next page)
Next, let us try out a finite non-invertible DDS:

```python
sage: D = DiscreteDynamicalSystem(tuple(range(13)), lambda x: (x**2) % 13)
sage: D.ground_set()
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
sage: D.evolution()(4)
3
sage: D.orbit(4)
[4, 3, 9]
sage: D.orbit(1)
[1]
sage: D.orbit(3)
[3, 9]
```

Note that the finiteness is automatically being inferred here, since the (finite) tuple `tuple(range(13))` has been provided as the ground set.

Finally, here is a finite invertible DDS:

```python
sage: X = cartesian_product([[0, 1]]*8)
sage: Y = [s for s in X if sum(s) == 4]
sage: rot = lambda s : s[1:] + (s[0],)
```

We could have just as well provided its inverse explicitly:

```python
sage: rot7 = lambda s: (s[-1],) + s[:-1]
sage: D = DiscreteDynamicalSystem(Y, rot, inverse=rot7)
sage: D.evolution()((0, 1, 0, 0, 1, 0, 0, 1))
(1, 1, 0, 0, 1, 0, 0, 1)
sage: D.inverse_evolution()((0, 1, 0, 0, 1, 0, 0, 1))
(1, 1, 0, 0, 1, 0, 0, 1)
```

evolution()  
Return the evolution of self.

EXAMPLES:

```python
sage: D = DiscreteDynamicalSystem([1, 3, 4], lambda x: (3 if x == 4 else 1), create_tuple=True)
sage: ev = D.evolution()
sage: ev(1)
1
sage: ev(4)
3
```

evolution_power(n)  
Return the $n$-th power (with respect to composition) of the evolution of self.

This requires $n$ to be a nonnegative integer.

EXAMPLES:

```python
sage: D = DiscreteDynamicalSystem(range(10), lambda x: (x + 3) % 10, create_tuple=True)
sage: ev3 = D.evolution_power(3)
sage: ev3(1)
0
sage: ev3(2)
1
sage: ev0 = D.evolution_power(0)
sage: ev0(1)
1
sage: ev0(2)
2
sage: D.evolution_power(-1)
Traceback (most recent call last):
  ... ValueError: the n-th power of evolution is only defined for nonnegative integers n
```

ground_set()  
Return the ground set of self.

This will return None if no ground set was provided in the construction of self.

**Warning:** Unless self has been constructed with the create_tuple parameter set to True, this method will return whatever ground set was provided to the constructor. In particular, if a list was provided, then this precise list will be returned; mutating this list will then corrupt self.

EXAMPLES:
Discrete dynamics, Release 10.1

\begin{verbatim}
sage: D = DiscreteDynamicalSystem([1, 3, 4], lambda x: (3 if x == 4 else 1), create_tuple=True)
sage: D.ground_set()
(1, 3, 4)

is_homomesic(h, average=None, find_average=False, elements=None)
Check if h (a map from the ground set of self to a Q-vector space) is homomesic with respect to self.
If the optional argument average is provided, then this also checks that the averages are equal to average.
If the optional argument find_average is set to True, then this method returns the average of h in case h
is homomesic (instead of returning True).
If the optional argument elements (an iterable of elements of the ground set of self) is provided, then
this method only checks homomesy for the cycles in the orbits of the elements given in the list elements.
Note that elements must be provided if the ground set of self is infinite (or cannot be iterated through
for any other reason), since there is no way to check all the cycles in this case.
This method will fail to terminate if any element of elements has an infinite orbit.

Let us recall the definition of homomesy: Let (X, \phi) be a DDS. A cycle of (X, \phi) is a finite list u =
(u_1, u_2, \ldots, u_k) of elements of X such that \phi(u_i) = u_{i+1} for each i \leq k, where we set u_{k+1} = u_1. Note
that any element of X whose orbit is finite has a cycle in its orbit. Now, let h be a map from X to a Q-vector
space A. If u = (u_1, u_2, \ldots, u_k) is any cycle of (X, \phi), then the average of h on this cycle is defined to be
the element (h(u_1) + h(u_2) + \cdots + h(u_k))/k of A. We say that h is homomesic (with respect to the DDS
(X, \phi)) if and only if the averages of h on all cycles of (X, \phi) are equal.

EXAMPLES:
\begin{verbatim}
sage: W = Words(2, 5)
sage: F = DiscreteDynamicalSystem(W, lambda x: x[1:] + Word([x[0]]), is_finite=True, inverse=True)
sage: F.is_homomesic(lambda w: sum(w))
False
sage: F.is_homomesic(lambda w: 1, average=1)
True
sage: F.is_homomesic(lambda w: 1, average=0)
False
sage: F.is_homomesic(lambda w: 1)
True
sage: F.is_homomesic(lambda w: 1, find_average=True)
1
sage: F.is_homomesic(lambda w: w[0] - w[1], average=0)
True
sage: F.is_homomesic(lambda w: w[0] - w[1], find_average=True)
0
\end{verbatim}
\end{verbatim}

Now, let us check homomesy restricted to specific cycles:
\begin{verbatim}
sage: F = finite_dynamical_systems.bitstring_rotation(7)
sage: descents = lambda x: sum(1 for i in range(6) if x[i] > x[i+1])
sage: F.is_homomesic(descents)
False
sage: F.is_homomesic(descents, elements=[(1, 0, 1, 0, 0, 0, 0), (1, 0, 0, 1, 0, 0, 0)])
True
\end{verbatim}
\end{verbatim}
(continues on next page)
And here is a non-invertible finite dynamical system:

```
sage: F = finite_dynamical_systems.one_line([9, 1, 1, 6, 5, 4, 5, 5, 1])
sage: F.is_homomesic(lambda i: i)
True
sage: F.is_homomesic(lambda i: i % 2)
False
sage: F.is_homomesic(lambda i: i % 2, elements=[2, 9, 7])
True
sage: F.is_homomesic(lambda i: i % 2, elements=[2, 9, 4])
False
sage: F.is_homomesic(lambda i: i % 2, elements=[2, 9, 5, 7, 8, 2])
True
```

```
orbit(x, preperiod=False)
```

Return the orbit of the element $x$ of the ground set of `self` under the evolution $\phi$ of `self`.

This orbit is a list beginning with $x$ and ending with the last element that is not a repetition of a previous element. If the orbit is infinite, then this method does not terminate!

If the optional argument `preperiod` is set to `True`, then this method returns a pair $(o, k)$, where $o$ is the orbit of `self`, while $k$ is the smallest nonnegative integer such that $\phi^k(x) \in \{\phi^i(x) \mid i > k\}$.

The orbit of the element $x$ is also called the “rho” of $x$, due to its shape when it is depicted as a directed graph.

**EXAMPLES:**

```
sage: D = DiscreteDynamicalSystem(tuple(range(11)), lambda x: (x ** 2) % 11)
sage: D.orbit(6)
[6, 3, 9, 4, 5]
sage: D.orbit(6, preperiod=True)
([6, 3, 9, 4, 5], 1)
sage: D.orbit(3)
[3, 9, 4, 5]
sage: D.orbit(3, preperiod=True)
([3, 9, 4, 5], 0)
sage: D.orbit(9)
[9, 4, 5, 3]
sage: D.orbit(0)
[0]
```
A finite discrete dynamical system (henceforth FDDS) is a pair \((X, \phi)\) of a finite set \(X\) and a map \(\phi : X \rightarrow X\). This set \(X\) is called the ground set of the FDDS, while the map \(\phi\) is called the evolution of the FDDS.

The ground set \(X\) should always be provided as an iterable when defining a :class:`FiniteDynamicalSystem`.

**EXAMPLES:**

```python
sage: from sage.dynamics.finite_dynamical_system import FiniteDynamicalSystem
sage: D = FiniteDynamicalSystem(tuple(range(11)), lambda x: (x**2) % 11)
sage: D.ground_set()
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
sage: D.evolution()(4)
5
sage: D.orbit(4)
[4, 5, 3, 9]
sage: D.orbit(1)
[1]
sage: D.orbit(2)
[2, 4, 5, 3, 9]
```

```
sage: X = cartesian_product([[0, 1]]*8)
sage: Y = [s for s in X if sum(s) == 4]
sage: rot = lambda s : s[1:] + (0,)
sage: D = FiniteDynamicalSystem(Y, rot)
sage: D.evolution()((1, 1, 1, 0, 1, 0, 0, 1))
(1, 1, 0, 1, 0, 0, 1, 0)
```

**cycles()**

Return a list of all cycles of self, up to cyclic rotation.

We recall the definition of cycles: Let \((X, \phi)\) be a DDS. A cycle of \((X, \phi)\) is a finite list \(u = (u_1, u_2, \ldots, u_k)\) of elements of \(X\) such that \(\phi(u_i) = u_{i+1}\) for each \(i \leq k\), where we set \(u_{k+1} = u_1\). Note that any element of \(X\) whose orbit is finite has a cycle in its orbit.

**EXAMPLES:**

```python
sage: BS = finite_dynamical_systems.bulgarian_solitaire
sage: BS(8).cycles()
[[[4, 3, 1], [3, 3, 2], [3, 2, 2, 1], [4, 2, 1, 1]],
 [[[4, 2, 2], [3, 3, 1, 1]]]
```

```python
sage: D = DiscreteDynamicalSystem(tuple(range(6)), lambda x: (x + 2) % 6)
sage: D.cycles()
[[[5, 1, 3], [4, 0, 2]]
```

```python
sage: D = DiscreteDynamicalSystem(tuple(range(6)), lambda x: (x ** 2) % 6)
sage: D.cycles()
[[[1], [4], [3], [0]]
```

```python
sage: D = DiscreteDynamicalSystem(tuple(range(11)), lambda x: (x ** 2 - 1) % 11)
sage: D.cycles()
[[[10, 0], [8], [4]]
```

```python
sage: F = finite_dynamical_systems.one_line([4, 7, 2, 6, 2, 10, 9, 11, 5, 6, 12, 12, 12, 6])
```

(continues on next page)
sage: F.cycles()
[[[6, 10], [12], [9, 5, 2, 7]]

\textbf{is_invariant}(f)

Check if \(f\) is an invariant of \(\text{self}\).

Let \((X, \phi)\) be a discrete dynamical system. Let \(Y\) be any set. Let \(f: X \to Y\) be any map. Then, we say that \(f\) is an \textit{invariant} of \((X, \phi)\) if and only if \(f \circ \phi = f\).

\textbf{EXAMPLES:}

\begin{verbatim}
sage: W = Words(2, 5)
sage: F = DiscreteDynamicalSystem(W, lambda x: x[1:] + Word([x[0]]), is_finite=True)
sage: F.is_invariant(lambda w: sum(w))
True
sage: F.is_invariant(lambda w: 1)
True
sage: F.is_invariant(lambda w: w[0] - w[1])
False
sage: F.is_invariant(lambda w: sum(i**2 for i in w))
True
\end{verbatim}

Invariants and non-invariants of a permutation:

\begin{verbatim}
sage: F = finite_dynamical_systems.permutation([3, 4, 5, 6, 1, 2])
sage: F.is_invariant(lambda i: i % 2)
True
sage: F.is_invariant(lambda i: i % 3)
False
sage: F.is_invariant(lambda i: i > 1)
False
sage: F.is_invariant(lambda i: i % 2 == 0)
True
\end{verbatim}

\textbf{class} \texttt{sage.dynamics.finite_dynamical_system.InvertibleDiscreteDynamicalSystem}(X, phi, inverse=None, cache_orbits=False, create_tuple=False)

Bases: \texttt{DiscreteDynamicalSystem}

An invertible discrete dynamical system.

A \textit{discrete dynamical system} (henceforth DDS) is a pair \((X, \phi)\) of a set \(X\) and a map \(\phi : X \to X\). This set \(X\) is called the \textit{ground set} of the DDS, while the map \(\phi\) is called the \textit{evolution} of the DDS. An \textit{invertible DDS} is a DDS \((X, \phi)\) whose evolution \(\phi\) is invertible. In that case, \(\phi^{-1}\) is called the \textit{inverse evolution} of the DDS.

See \texttt{DiscreteDynamicalSystem} for details.

\textbf{INPUT:}

- \(X\) – set, list, tuple, or another iterable, or \texttt{None}; the ground set for the DDS. This can be \texttt{None} in case of a \texttt{DiscreteDynamicalSystem} or a \texttt{InvertibleDiscreteDynamicalSystem}. Make sure to set the create_tuple argument to \texttt{True} if you provide an iterator or a list for \(X\), as otherwise the input would be exposed.
• phi – function, or callable that acts like a function; the evolution of the DDS.

• inverse – function, or callable that acts like a function; the inverse evolution of the DDS. (A default implementation is implemented when this argument is not provided; but it assumes the orbits to be finite.)

• cache_orbits – boolean (default: False); whether or not the orbits should be cached once they are computed.

• create_tuple – boolean (default: False); whether or not the input X should be translated into a tuple (set this to True to prevent mutation if X is a list, and to prevent exhaustion if X is an iterator).

EXAMPLES:

```python
sage: from sage.dynamics.finite_dynamical_system import InvertibleDiscreteDynamicalSystem
sage: D = InvertibleDiscreteDynamicalSystem(NN, lambda x: (x + 2 if x % 4 < 2 else x - 2))
sage: D.ground_set()
Non negative integer semiring
sage: D.evolution()(5)
7
sage: D.evolution()(6)
4
sage: D.evolution()(4)
6
sage: D.inverse_evolution()(4)
6
```

The necessity of create_tuple=True:

```python
sage: X = [0, 1, 2, 3, 4]
sage: D_wrong = InvertibleDiscreteDynamicalSystem(X, lambda x: (x**3) % 5)
sage: D_right = InvertibleDiscreteDynamicalSystem(X, lambda x: (x**3) % 5, create_tuple=True)
```

```python
sage: X[4] = 666 # evil
sage: D_wrong.ground_set()
[0, 1, 2, 3, 666]
sage: D_right.ground_set()
(0, 1, 2, 3, 4)
```

**evolution_power**(n)

Return the n-th power (with respect to composition) of the evolution of self.

This requires n to be an integer.

EXAMPLES:

```python
sage: D = DiscreteDynamicalSystem(range(10), lambda x: (x + 3) % 10, create_tuple=True, inverse=True)
sage: ev3 = D.evolution_power(3)
sage: ev3(1)
0
sage: ev3(2)
1
sage: ev0 = D.evolution_power(0)
sage: ev0(1)
1
```

(continues on next page)
sage: ev0(2)
2
sage: evm1 = D.evolution_power(-1)
sage: evm1(1)
8
sage: evm1(2)
9
sage: evm2 = D.evolution_power(-2)
sage: evm2(1)
5
sage: evm2(2)
6

inverse_evolution()
Return the inverse evolution of self (as a map from the ground set of self to itself).

EXAMPLES:

sage: D = DiscreteDynamicalSystem(tuple(range(8)), lambda x: (x + 2) % 8, inverse=True)
sage: D.inverse_evolution()(1)
7
sage: D.inverse_evolution()(3)
1

sage: D = DiscreteDynamicalSystem(ZZ, lambda x: (x + 2) % 8, inverse=True)
sage: D.inverse_evolution()(1)
7
sage: D.inverse_evolution()(3)
1

inverse_evolution_default(x)
Return the inverse evolution of self, applied to the element x of the ground set of self.

This is the default implementation, assuming that the orbit of x is finite.

EXAMPLES:

sage: D = DiscreteDynamicalSystem(tuple(range(8)), lambda x: (x + 2) % 8, inverse=True)
sage: D.inverse_evolution_default(1)
7
sage: D.inverse_evolution_default(3)
1

sage: D = DiscreteDynamicalSystem(ZZ, lambda x: (x + 2) % 8, inverse=True)
sage: D.inverse_evolution_default(1)
7
sage: D.inverse_evolution_default(3)
1

orbit(x, preperiod=False)
Return the orbit of the element x of the ground set of self.

This orbit is a list beginning with x and ending with the last element until x reappears. If x never reappears, then this will not terminate!

If the optional argument preperiod is set to True, then this method returns a pair (o, k), where o is the orbit of self, while k is the smallest nonnegative integer such that $\phi^k(x) \in \{\phi^i(x) \mid i > k\}$. Note that k is necessarily 0, since the DDS self is invertible!

EXAMPLES:
Discrete dynamics, Release 10.1

```python
sage: D = DiscreteDynamicalSystem(tuple(range(8)), lambda x: (x + 2) % 8, inverse=True)

sage: D.ground_set()
(0, 1, 2, 3, 4, 5, 6, 7)

sage: D.orbit(2)
[2, 4, 6, 0]

sage: D.orbit(5)
[5, 7, 1, 3]

sage: D.orbit(5, preperiod=True)
([5, 7, 1, 3], 0)

sage: D = DiscreteDynamicalSystem(ZZ, lambda x: (x + 2) % 8, inverse=True)

sage: D.ground_set()
Integer Ring

sage: D.orbit(2)
[2, 4, 6, 0]

sage: D.orbit(5)
[5, 7, 1, 3]

sage: D.orbit(5, preperiod=True)
([5, 7, 1, 3], 0)
```

**verify_inverse_evolution**(*x=None*)

Verify that the composition of evolution and inverse evolution on self is the identity (both ways).

The optional argument *x*, if provided, restricts the testing to the element *x* only. Otherwise, all elements of the ground set are tested (if they can be enumerated).

This is mostly used to check the correctness of self-implemented inverse evolution methods.

**EXAMPLES:**

```python
sage: D = DiscreteDynamicalSystem(tuple(range(8)), lambda x: (x + 2) % 8, inverse=True)

sage: D.verify_inverse_evolution()  # True

sage: D.verify_inverse_evolution(3)  # True

sage: fake_inverse = lambda x: x

sage: D = DiscreteDynamicalSystem(tuple(range(8)), lambda x: (x + 2) % 8, inverse=fake_inverse)

sage: D.verify_inverse_evolution()  # False

sage: D.verify_inverse_evolution(3)  # False
```

**class** `sage.dynamics.finite_dynamical_system.InvertibleFiniteDynamicalSystem`

An invertible finite discrete dynamical system.

A *finite discrete dynamical system* (henceforth *FDDS*) is a pair (*X*, *φ*) of a finite set *X* and a map *φ*: *X* → *X*. This set *X* is called the *ground set* of the FDDS, while the map *φ* is called the *evolution* of the FDDS. An FDDS...
(X, φ) is called invertible if the map φ is invertible; in this case, φ⁻¹ is called the inverse evolution of the FDDS. The ground set X should always be provided as an iterable when defining a FiniteDynamicalSystem.

EXAMPLES:

```
sage: from sage.dynamics.finite_dynamical_system import InvertibleFiniteDynamicalSystem
c
sage: D = InvertibleFiniteDynamicalSystem(tuple(range(5)), lambda x: (x + 2) % 5)
sage: D.ground_set()
(0, 1, 2, 3, 4)
sage: D.evolution()(4)
1
sage: D.orbits()
[[4, 1, 3, 0, 2]]
sage: D.inverse_evolution()(2)
0
sage: D.inverse_evolution()(1)
4
sage: D.evolution_power(-1)(1)
4
sage: D.evolution_power(-2)(1)
2
sage: X = cartesian_product([[0, 1]]*8)
sage: Y = [s for s in X if sum(s) == 4]
sage: rot = lambda s : s[1:] + (s[0],)
sage: D = InvertibleFiniteDynamicalSystem(Y, rot)
sage: D.evolution()((0, 1, 1, 0, 1, 0, 0, 1))
(1, 1, 0, 1, 0, 0, 1, 0)
sage: D.inverse_evolution()((0, 1, 1, 0, 1, 0, 0, 1))
(1, 0, 1, 1, 0, 1, 0, 0)
sage: sorted(D.orbit_lengths())
[2, 4, 8, 8, 8, 8, 8, 8, 8, 8]
```

cycles()
Return a list of all cycles of self, up to cyclic rotation.

We recall the definition of cycles: Let (X, φ) be a DDS. A cycle of (X, φ) is a finite list u = (u₁, u₂, ..., uₖ) of elements of X such that φ(uᵢ) = uᵢ₊₁ for each i ≤ k, where we set uₖ₊₁ = u₁. Note that any element of X whose orbit is finite has a cycle in its orbit.

Since self is invertible, the cycles of self are the same as its orbits.

EXAMPLES:

```
sage: D = DiscreteDynamicalSystem(tuple(range(6)), lambda x: (x + 2) % 6, inverse=True)
sage: D.cycles()
[[5, 1, 3], [4, 0, 2]]
sage: D = DiscreteDynamicalSystem(tuple(range(6)), lambda x: (x + 3) % 6, inverse=True)
sage: D.cycles()
[[5, 2], [4, 1], [3, 0]]
```

orbit_lengths()
Return a list of the lengths of all orbits of self.
EXAMPLES:

```python
sage: D = DiscreteDynamicalSystem(tuple(range(6)), lambda x: (x + 2) % 6,
            inverse=True)
sage: D.orbit_lengths()
[3, 3]
sage: D = DiscreteDynamicalSystem(tuple(range(6)), lambda x: (x + 3) % 6,
            inverse=True)
sage: D.orbit_lengths()
[2, 2, 2]
```

**orbits()**

Return a list of all orbits of self, up to cyclic rotation.

EXAMPLES:

```python
sage: D = DiscreteDynamicalSystem(tuple(range(6)), lambda x: (x + 2) % 6,
            inverse=True)
sage: D.orbits()
[[5, 1, 3], [4, 0, 2]]
sage: D = DiscreteDynamicalSystem(tuple(range(6)), lambda x: (x + 3) % 6,
            inverse=True)
sage: D.orbits()
[[5, 2], [4, 1], [3, 0]]
```
SANDPILES

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.
Discrete dynamics, Release 10.1

- 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:
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])
c.equivalent_recurrent()
```

```
{1: 2, 2: 2, 3: 2, 4: 0}
```

Some group operations:

```sage
m = S.max_stable()
i = S.identity()
m.values()
{2, 2, 2, 1}
i.values()
{2, 2, 2, 0}
m + i  # coordinate-wise sum
{1: 4, 2: 4, 3: 4, 4: 1}
i.values()
{1: 0, 2: 0, 3: 0, 4: 1}
m - i
{1: 0, 2: 2, 3: 0, 4: 1}
m & i # add, then stabilize
{1: 2, 2: 2, 3: 2, 4: 1}
e = m + m
```
Discrete dynamics, Release 10.1

{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}
sage: a^3     # a*a*a
{1: 2, 2: 2, 3: 2, 4: 1}
sage: a^(-1) == m
True
sage: a < m  # every coordinate of a is < that of m
True

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:
and its Betti numbers:

\[ \begin{array}{cccccc}
0: & 1 & 1 & - & - & - \\
1: & - & 2 & 2 & - & - \\
2: & - & 4 & 13 & 13 & 4 \\
\hline
\text{total:} & 1 & 7 & 15 & 13 & 4
\end{array} \]

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 [0..max(a)] if a.count(i)])`
- `sage: t = text("Distribution of avalanche sizes", (2,2), rgbcolor=(1,0,0))`
- `sage: show(p+t, axes_labels=['log(N)', 'log(D(N))'])`

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)`

(continues on next page)
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
e
sage: D.q_reduced()
{0: 4, 1: 0, 2: 0, 3: 1}
sage: D.rank()
2
e
sage: sorted(D.effective_div(), key=str)
[{0: 0, 1: 0, 2: 0, 3: 5},
 {0: 0, 1: 0, 2: 4, 3: 1},
 {0: 0, 1: 4, 2: 0, 3: 1},
 {0: 1, 1: 1, 2: 1, 3: 2},
 {0: 4, 1: 0, 2: 0, 3: 1}]
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]]
e
sage: D.rank()
2
e
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
e
sage: rank(D - E)
-1
e
sage: (D - E).effective_div()
[]
e
sage: D.weierstrass_pts()
(0, 1, 2, 3)
e
sage: D.weierstrass_rank_seq(0)
(2, 1, 0, 0, -1)
e
sage: D.weierstrass_pts()
(0, 1, 2, 3)
e
sage: D.weierstrass_rank_seq(0)
(2, 1, 0, 0, -1)
e

class sage.sandpiles.sandpile.Sandpile(g, sink=None)
Bases: DiGraph
Class for Dhar’s abelian sandpile model.
e
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

**EXAMPLES:**

```
sage: S = sandpiles.House()
sage: S.all_k_div(7)
{0: 7, 1: 7, 2: 7, 3: 7, 4: 7}
```

### avalanche_polynomial\((\text{multivariable}=\text{True})\)

The avalanche polynomial. See NOTE for details.

**INPUT:**

\(\text{multivariable} – (\text{default}: \text{True})\) 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 recurrernts and \(v\) ranges over the nonsink vertices. If \(\text{multivariable}\) is \(\text{False}\), 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\((\text{verbose}=\text{True})\)

The Betti table for the homogeneous toppling ideal. If \(\text{verbose}\) is \(\text{True}\), it prints the standard Betti table, otherwise, it returns a less formatted table.

**INPUT:**

\(\text{verbose} – (\text{default}: \text{True})\) boolean

**OUTPUT:**

Betti numbers for the sandpile
EXAMPLES:

```python
sage: S = sandpiles.Diamond()
sage: S.betti()
```

```
0   1   2   3
--- --- --- ---
0: 1   -   -   -
1: -   2   -   -
2: -   4   9   4
--- --- --- ---
total: 1   6   9   4
```

```python
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
```

```python
sage: len(p)
11
sage: p[0][1].homology()
{0: Z, 1: 0}
sage: p[-1][1].homology()
{0: 0, 1: 0, 2: Z}
```

**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:**
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: 2, 3: 2, 4: 1, 5: 4}
sage: script = S.burning_script().values()
sage: script
[1, 2, 2, 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 $L$ is the reduced Laplacian, then $\sigma \cdot L = b$.

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 $\sigma$.

b\n

burning_script()

A script for the minimal burning configuration.

OUTPUT:

dict

EXAMPLES:

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: 2, 3: 2, 4: 1, 5: 4}
sage: script = S.burning_script().values()
sage: script
[1, 2, 2, 1, 4]

(continues on next page)
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)
sage: S.canonical_divisor()
{0: 1, 1: 1, 2: 1, 3: 1}
sage: s = Sandpile({0:[1,1],1:[0,0,1,1,1]},0)
sage: s.canonical_divisor()  # loops are disregarded
{0: 0, 1: 0}
```

Warning: The underlying graph must be undirected.

### dict()

A dictionary of dictionaries representing a directed graph.

OUTPUT:

dict

EXAMPLES:

```
sage: S = sandpiles.Diamond()
sage: S.dict()
{0: {1: 1, 2: 1},
  1: {0: 1, 2: 1, 3: 1},
  2: {0: 1, 1: 1, 3: 1},
```

(continues on next page)
3: {1: 1, 2: 1}
sage: S.sink()
0

**genus()**
The genus: (# non-loop edges) - (# vertices) + 1. Only defined for undirected graphs.

**OUTPUT:**
integer

**EXAMPLES:**

```sage
sage: sandpiles.Complete(4).genus()
3
sage: sandpiles.Cycle(5).genus()
1
```

**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:**

```sage
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:**

```sage
sage: s = sandpiles.Cycle(5)
sage: s.group_gens()
[{1: 0, 2: 1, 3: 1, 4: 1}]
sage: s.group_gens()[0].order()
5
sage: s = sandpiles.Complete(5)
sage: s.group_gens(False)
[[2, 3, 2, 2], [2, 2, 3, 2], [2, 2, 2, 3]]
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:

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:

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(\texttt{\text敔se=\texttt{True}})
List of Sandpile-specific methods (not inherited from \texttt{Graph}). If \texttt{\text蓏se}, include short descriptions.

INPUT:
\texttt{\text㮾se=\texttt{True}} -- (default: \texttt{True}) boolean

OUTPUT:
printed string

EXAMPLES:

sage: Sandpile.help()  # long time
For detailed help with any method \texttt{FOO} listed below, enter \texttt{"Sandpile.FOO?"} or enter \texttt{"S.FOO?"} for any Sandpile \texttt{S}.

all_k_config -- The constant configuration with all values set to \texttt{k}.
all_k_div -- The divisor with all values set to \texttt{k}.
avanche_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.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dict</td>
<td>A dictionary of dictionaries representing a directed graph.</td>
</tr>
<tr>
<td>genus</td>
<td>The genus: (# non-loop edges) - (# vertices) + 1.</td>
</tr>
<tr>
<td>groebner</td>
<td>A Groebner basis for the homogeneous toppling ideal.</td>
</tr>
<tr>
<td>group_gens</td>
<td>A minimal list of generators for the sandpile group.</td>
</tr>
<tr>
<td>group_order</td>
<td>The size of the sandpile group.</td>
</tr>
<tr>
<td>h_vector</td>
<td>The number of superstable configurations in each degree.</td>
</tr>
<tr>
<td>help</td>
<td>List of Sandpile-specific methods (not inherited from ...Graph...).</td>
</tr>
<tr>
<td>hilbert_function</td>
<td>The Hilbert function of the homogeneous toppling ideal.</td>
</tr>
<tr>
<td>ideal</td>
<td>The saturated homogeneous toppling ideal.</td>
</tr>
<tr>
<td>identity</td>
<td>The identity configuration.</td>
</tr>
<tr>
<td>in_degree</td>
<td>The in-degree of a vertex or a list of all in-degrees.</td>
</tr>
<tr>
<td>invariant_factors</td>
<td>The invariant factors of the sandpile group.</td>
</tr>
<tr>
<td>is_undirected</td>
<td>Is the underlying graph undirected?</td>
</tr>
<tr>
<td>jacobian_representatives</td>
<td>Representatives for the elements of the Jacobian group.</td>
</tr>
<tr>
<td>laplacian</td>
<td>The Laplacian matrix of the graph.</td>
</tr>
<tr>
<td>markov_chain</td>
<td>The sandpile Markov chain for configurations or divisors.</td>
</tr>
<tr>
<td>max_stable</td>
<td>The maximal stable configuration.</td>
</tr>
<tr>
<td>max_stable_div</td>
<td>The maximal stable divisor.</td>
</tr>
<tr>
<td>max_superstables</td>
<td>The maximal superstable configurations.</td>
</tr>
<tr>
<td>min_recurrents</td>
<td>The minimal recurrent elements.</td>
</tr>
<tr>
<td>nonsink_vertices</td>
<td>The nonsink vertices.</td>
</tr>
<tr>
<td>nonspecial_divisors</td>
<td>The nonspecial divisors.</td>
</tr>
<tr>
<td>out_degree</td>
<td>The out-degree of a vertex or a list of all out-degrees.</td>
</tr>
<tr>
<td>picard_representatives</td>
<td>Representatives of the divisor classes of degree d in the Picard group.</td>
</tr>
<tr>
<td>points</td>
<td>Generators for the multiplicative group of zeros of the sandpile ideal.</td>
</tr>
<tr>
<td>postulation</td>
<td>The postulation number of the toppling ideal.</td>
</tr>
<tr>
<td>recurrences</td>
<td>The recurrent configurations.</td>
</tr>
<tr>
<td>reduced_laplacian</td>
<td>The reduced Laplacian matrix of the graph.</td>
</tr>
<tr>
<td>reorder_vertices</td>
<td>A copy of the sandpile with vertex names permuted.</td>
</tr>
<tr>
<td>resolution</td>
<td>A minimal free resolution of the homogeneous toppling ideal.</td>
</tr>
<tr>
<td>ring</td>
<td>The ring containing the homogeneous toppling ideal.</td>
</tr>
<tr>
<td>show</td>
<td>Draw the underlying graph.</td>
</tr>
<tr>
<td>show3d</td>
<td>Draw the underlying graph.</td>
</tr>
<tr>
<td>sink</td>
<td>The sink vertex.</td>
</tr>
<tr>
<td>smith_form</td>
<td>The Smith normal form for the Laplacian.</td>
</tr>
<tr>
<td>solve</td>
<td>Approximations of the complex affine zeros of the sandpile ideal.</td>
</tr>
<tr>
<td>stable_configs</td>
<td>Generator for all stable configurations.</td>
</tr>
<tr>
<td>stationary_density</td>
<td>The stationary density of the sandpile.</td>
</tr>
<tr>
<td>superstables</td>
<td>The superstable configurations.</td>
</tr>
<tr>
<td>symmetric_recurrents</td>
<td>The symmetric recurrent configurations.</td>
</tr>
</tbody>
</table>
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)
sage: s.hilbert_function()
[1, 5, 15, 31, 45]
sage: 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()
sage: S.ideal()  # 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(verbos=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:**

```python
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()

Let \( \tau \) be the nonnegative generator of the kernel of the transpose of the Laplacian, and let \( \tau_s \) be its 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.

\textbf{laplacian}() 

The Laplacian matrix of the graph. Its \textit{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
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, 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
[2, 2, 1]
sage: m = s.markov_chain(s.zero_div(), [0.1,0.1,0.1,0.7])
sage: next(m).values()  # random
[0, 0, 0, 1]
sage: next(m).values()  # random
[0, 0, 1, 1]
sage: next(m).values()  # random
[0, 0, 1, 2]
sage: next(m).values()  # random
[1, 1, 2, 0]
sage: next(m).values()  # random
[1, 1, 2, 1]
sage: next(m).values()  # random
[1, 1, 2, 2]
sage: next(m).values()  # random
[1, 1, 2, 3]
sage: next(m).values()  # random
[1, 1, 2, 4]
```
sage: next(m).values() # random
[1, 1, 3, 4]

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 \texttt{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 \texttt{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 \texttt{distrib} is specified, its length is equal to the total number of vertices (including the sink).

REFERENCES:

• [Lev2014]

\textbf{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\}

\textbf{max\_stable\_div()}

The maximal stable divisor.

OUTPUT:

SandpileDivisor (the maximal stable divisor)

EXAMPLES:

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\}

\textbf{max\_superstables(\texttt{verbose=\text{True}})}

The maximal superstable configurations. If the underlying graph is undirected, these are the superstables of highest degree. If \texttt{verbose} is \texttt{False}, the configurations are converted to lists of integers.

INPUT:

\texttt{verbose} – (default: \texttt{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.superstables(True)
[[1, 0, 1], [0, 2, 0], [2, 0, 0], [0, 1, 1]]
sage: s.h_vector()
[1, 3, 4]
```

`min_recurrents(verbos=False)`

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]]
sage: s.min_recurrents(False)
[[1, 2, 0], [2, 0, 1], [0, 2, 1], [2, 1, 0]]
sage: [i.deg() for i in s.recurrents()]
[5, 4, 3, 3, 3, 3, 4, 4]
```

`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)]

\textbf{nonspecial\_divisors}(\texttt{verbose=\textit{True}})

The nonspecial divisors. Only for undirected graphs. (See NOTE.)

INPUT:

\texttt{verbose} – (default: \textit{True}) boolean

OUTPUT:

list (of divisors)

EXAMPLES:

\begin{verbatim}
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]
[[], [], [], [], [], []]
\end{verbatim}

\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 \textit{False}, the divisors are converted to lists of integers.

\textbf{Warning:} The underlying graph must be undirected.

\textbf{out\_degree}(\texttt{v=\textit{None}})

The out-degree of a vertex or a list of all out-degrees.

INPUT:

\begin{verbatim}
v - (optional) vertex name
\end{verbatim}

OUTPUT:

integer or dict

EXAMPLES:

\begin{verbatim}
sage: s = sandpiles.House()
sage: s.out\_degree()
\{0: 2, 1: 2, 2: 3, 3: 3, 4: 2\}
sage: s.out\_degree(2)
3
\end{verbatim}

\textbf{picard\_representatives}(\texttt{d}, \texttt{verbose=\textit{True}})

Representatives of the divisor classes of degree \( d \) in the Picard group. (Also see the documentation for \texttt{jacobian\_representatives}.)
INPUT:

• d − integer
• verbose – (default: True) boolean

OUTPUT:

list of SandpileDivisors (or lists representing divisors)

EXAMPLES:

```
sage: s = sandpiles.Complete(3)
sage: s.superstables(False)
[[0, 0], [0, 1], [1, 0]]
sage: s.jacobian_repsrepresentatives(False)
[[0, 0, 0], [-1, 0, 1], [-1, 1, 0]]
sage: s.picard_repsrepresentatives(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.

```
sage: S = sandpiles.Complete(4)
sage: S.points()
[[-I, I, 1], [-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:

```
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:
Discrete dynamics, Release 10.1

```
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],
 [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.

OUTPUT:

Sandpile

EXAMPLES:
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:

sage: T.dict()
{0: {1: 1}, 1: {0: 1, 2: 1}, 2: {0: 1}}

resolution(\texttt{verbose=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:

two lists of matrices representing the mappings of the free resolution of the toppling ideal

EXAMPLES:

sage: S = Sandpile({0: {}, 1: {0: 1, 2: 1, 3: 4}, 2: {3: 5}, 3: {1: 1, 2: 1}}, 0)
sage: S.resolution()
\texttt{# a Gorenstein sandpile graph}
'R^(1) \leftarrow R^(5) \leftarrow R^(5) \leftarrow R^(1)'
sage: S.resolution(\texttt{True})
\[
\begin{bmatrix}
x^1^2 - x^3^*x^0 & x^3^*x^1 & x^2^*x^0 & x^3^2 & x^2^*x^1 & x^2^*x^3 & x^0^2 & x^2^2 & x^1^*x^0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]
sage: r = S.resolution(\texttt{True})
sage: r[0]*r[1]
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
sage: r[1]*r[2]
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}

ring()
The ring containing the homogeneous toppling ideal.

OUTPUT:

ring

EXAMPLES:
Discrete dynamics, Release 10.1

```
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 by 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: 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: S = sandpiles.House()
sage: S.show3d() # long time
```

**sink()**

The sink vertex.

**OUTPUT:**

sink vertex

**EXAMPLES:**

```
sage: G = sandpiles.House()
sage: G.sink()
@
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()
[[-0.707107000000000 + 0.707107000000000*I,
  0.707107000000000 - 0.707107000000000*I],
 [-0.707107000000000 - 0.707107000000000*I,
  0.707107000000000 + 0.707107000000000*I],
 [-I, -I],
 [I, I],
 [0.707107000000000 + 0.707107000000000*I,
 -0.707107000000000 - 0.707107000000000*I],
[0.707107000000000 - 0.707107000000000*I,
 -0.707107000000000 + 0.707107000000000*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:
gegenerator 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])
sage: list(b)
[{1: 0, 2: 0}, {1: 1, 2: 0}]
```

**stationary_density()**
The stationary density of the sandpile.

OUTPUT:
rational number

EXAMPLES:

```
sage: s = sandpiles.Complete(3)
sage: s.stationary_density()
10/9
sage: s = Sandpile(digraphs.DeBruijn(2,2),'00')
sage: s.stationary_density()
9/8
```

**Note:** 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(verbosetrue)**
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:
verbosetrue – (default: True) boolean

OUTPUT:
list of SandpileConfig

EXAMPLES:

```
sage: sp = Sandpile(graphs.HouseXGraph(),0).superstables()
sage: sp[:3]
[{1: 0, 2: 0, 3: 0, 4: 0}, {1: 1, 2: 0, 3: 0, 4: 1}, {1: 1, 2: 0, 3: 0, 4: 0}]
sage: sandpiles.Complete(4).superstables(False)
[[0, 0, 0],
```

(continues on next page)
Discrete dynamics, Release 10.1

(continued from previous page)

\[
\begin{align*}
&[0, 0, 1], \\
&[0, 1, 0], \\
&[1, 0, 0], \\
&[0, 0, 2], \\
&[0, 2, 0], \\
&[2, 0, 0], \\
&[0, 1, 1], \\
&[1, 0, 1], \\
&[1, 1, 0], \\
&[0, 1, 2], \\
&[0, 2, 1], \\
&[1, 0, 2], \\
&[1, 2, 0], \\
&[2, 0, 1], \\
&[2, 1, 0]
\end{align*}
\]

\text{sage: \texttt{sandpiles.Cycle(4).superstables(False)}}

\[
[[0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]]
\]

\text{symmetric\_recurrents}(orbits)

The symmetric recurrent configurations.

INPUT:

orbits - list of lists partitioning the vertices

OUTPUT:

list of recurrent configurations

EXAMPLES:

\text{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}})
\text{sage: S.symmetric\_recurrents([[1],[2,3],[4]])}
[[[1: 2, 2: 2, 3: 2, 4: 1}, \{1: 2, 2: 2, 3: 2, 4: 0}\]
\text{sage: S.recurrents()}
[[[1: 2, 2: 2, 3: 2, 4: 1}, \{1: 2, 2: 2, 3: 2, 4: 0}\]
\text{Note:} The user is responsible for ensuring that the list of orbits comes from a group of symmetries of the underlying graph.

\text{tutte\_polynomial()}

The Tutte polynomial of the underlying graph. Only defined for undirected sandpile graphs.
OUTPUT:

polynomial

EXAMPLES:

```
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.

OUTPUT:

ideal

EXAMPLES:

```
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^*x2 - 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:

```
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:

```
sage: s = sandpiles.Diamond()
sage: s.zero_config()
{1: 0, 2: 0, 3: 0}
```
zero_div()
The all-zero divisor.

OUTPUT:
SandpileDivisor

EXAMPLES:

```
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: 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: 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))'])  # long time
```
Note: If \texttt{distrib} is \texttt{None}, then the probability is the uniform probability on the nonsink vertices. Otherwise, there are two possibilities:

(i) the length of \texttt{distrib} is equal to the number of vertices, and \texttt{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 \texttt{distrib} must be equal to the number of nonsink vertices, and \texttt{distrib} represents a probability distribution on the nonsink vertices.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Warning:} If \texttt{distrib} $\neq \texttt{None}$, 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. \tabularnewline
\hline
\end{tabular}
\end{center}

\texttt{burst\_size}(\texttt{v})

The burst size of the configuration with respect to the given vertex.

INPUT:
\texttt{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 \texttt{c.burst(v)}, if \texttt{v} is not the sink, let \texttt{c'} be the unique recurrent for which the stabilization of \texttt{c' + v} is \texttt{c}. The burst size is then the amount of sand that goes into the sink during this stabilization. If \texttt{v} is the sink, the burst size is defined to be 1.

REFERENCES:

- [Lev2014]

\texttt{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: 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[v] for v in S.nonsink_vertices()])
sage: f = vector([x[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: 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
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
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(\texttt{verbose=True})

List of \texttt{SandpileConfig} methods. If \texttt{verbose}, include short descriptions.

INPUT:

\texttt{verbose} – (default: \texttt{True}) boolean

OUTPUT:

printed string

EXAMPLES:

```
sage: SandpileConfig.help()
Shortcuts for \texttt{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 \texttt{FOO} listed below, enter "SandpileConfig.FOO?" or enter "c.FOO?" for any \texttt{SandpileConfig} \texttt{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 \texttt{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.
```

(continues on next page)
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
sage: (S.max_stable() & S.max_stable()).is_stable()
True
```

is_superstable()
Is the configuration superstable?
OUTPUT:
boolean
EXAMPLES:

```
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:

```sage
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:**

```sage
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
sage: c.is_recurrent()
False
sage: c.equivalent_recurrent() == S.identity()
True
```

`sandpile()`:
The configuration’s underlying sandpile.

**OUTPUT:**
Sandpile

**EXAMPLES:**

```sage
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()
sage: '_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
sage: ~c == c.stabilize()
True
```

**support()**
The vertices containing sand.

**OUTPUT:**

list - support of the configuration

EXAMPLES:

```python
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
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
boolean

EXAMPLES:

```sage
sage: S = Sandpile({'a':[c,b], 'b':[c,a], 'c':[a]}, 'a')
sage: c = SandpileConfig(S, {'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
sage: S = sandpiles.House()
sage: p = SandpileDivisor(S, [1,2,1,0,0]).Dcomplex()
sage: p.homology()
{0: 0, 1: Z x 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.

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:

```python
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(\text{verbose=True}, \text{with_firing_vectors=False})

All linearly equivalent effective divisors. If \text{verbose} is \text{False}, the divisors are converted to lists of integers. If \text{with_firing_vectors} is \text{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:

- \text{verbose} -- (default: True) boolean
- \text{with_firing_vectors} -- (default: False) boolean

OUTPUT:

list (of divisors)

EXAMPLES:

```python
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, -1, -1, -2)),
 ({0: 0, 1: 2, 2: 4, 3: 0}, (0, -1, -2, -1)),
 ({0: 0, 1: 6, 2: 0, 3: 0}, (0, -2, -1, -1)),
 ({0: 1, 1: 3, 2: 1, 3: 1}, (0, -1, -1, -1)),
 ({0: 2, 1: 0, 2: 2, 3: 2}, (0, 0, -1, -1)),
 ({0: 4, 1: 2, 2: 0, 3: 0}, (0, 0, 0, 0))]
sage: a = _[2]
sage: a[0].values()
[0, 6, 0, 0]
sage: vector(D.values()) - s.laplacian()^a[1]
```
(0, 6, 0, 0)
\begin{verbatim}
sage: sorted(D.effective_div(False, True))
[([0, 2, 0, 4], (0, -1, -1, -2)),
 ([0, 2, 4, 0], (0, -1, -2, -1)),
 ([0, 6, 0, 0], (0, -2, -1, -1)),
 ([1, 3, 1, 1], (0, -1, -1, -1)),
 ([2, 0, 2, 2], (0, 0, -1, -1)),
 ([4, 2, 0, 0], (0, 0, 0, 0))]
sage: D = SandpileDivisor(s,[-1,0,0,0])
sage: D.effective_div(False,True)
[]
\end{verbatim}

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:

\begin{verbatim}
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
\end{verbatim}

fire_unstable()
Fire all unstable vertices.

OUTPUT:
SandpileDivisor

EXAMPLES:

\begin{verbatim}
sage: S = sandpiles.Cycle(3)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.fire_unstable()
{0: 3, 1: 1, 2: 2}
\end{verbatim}

fire_vertex($v$)
Fire the given vertex.

INPUT:

$v$ – vertex

OUTPUT:
SandpileDivisor

EXAMPLES:

```python
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(verbosetrue)

List of SandpileDivisor methods. If verbose, include short descriptions.

INPUT:

verbose – (default: True) boolean

OUTPUT:

printed string

EXAMPLES:

```python
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.
fine_unstable -- Fire all unstable vertices.
fine_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.
rank -- The rank of the divisor.
sandpile -- The divisor's underlying sandpile.
show -- Show the divisor.
simulate_threshold -- The first unstabilizable divisor in the closed Markov chain.
stabilize -- The stabilization of the divisor.
support -- List of vertices at which the divisor is nonzero.
unstable -- The unstable vertices.
values -- The values of the divisor as a list.
weierstrass_div -- The Weierstrass divisor.
weierstrass_gap_seq -- The Weierstrass gap sequence at the given vertex.
weierstrass_pts -- The Weierstrass points (vertices).
weierstrass_rank_seq -- The Weierstrass rank sequence at the given vertex.

93
is_alive(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, 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)
(0, -1, -1)
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)
()
```

Note:

• If with_firing_vector is False, returns either True or False.

• If with_firing_vector is True then: (i) if self is linearly equivalent to D, returns a vector v such that self - v*self.laplacian().transpose() = D. Otherwise, (ii) if self is not linearly equivalent to D, the output is the empty vector, ()

Chapter 4. Sandpiles
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:

```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:

```sage
s = Sandpile({'0': [1], '1': [1, 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 \( \text{orbits} \).

INPUT:

orbits – list of lists of vertices

OUTPUT:

boolean

EXAMPLES:

```sage
S = sandpiles.House()
sage: S.dict()
{0: {1: 1, 2: 1},
  1: {0: 1, 3: 1},
  2: {0: 1, 3: 1, 4: 1},
  3: {1: 1, 2: 1, 4: 1},
  4: {2: 1, 3: 1}}
sage: D = SandpileDivisor(S, [0,0,1,1,3])
sage: D.is_symmetric([[2,3], [4]])
True
```
is_weierstrass_pt(v='sink')
Is the given vertex a Weierstrass point?
INPUT:
v – (default: sink) vertex
OUTPUT:
boolean
EXAMPLES:
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

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 \( r(D), r(D) - 1, \ldots, 0, -1, -1, \ldots \)

polytope()
The polytope determining the complete linear system.
OUTPUT:
polytope
EXAMPLES:
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

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 \( x_{sink vertex} = 0 \). 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
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()
()
```

```python
q_reduced(\texttt{verbose=True})
The linearly equivalent \(q\)-reduced divisor.

INPUT:
verbose – (default: True) boolean

OUTPUT:
SandpileDivisor or list representing SandpileDivisor

EXAMPLES:

```sage
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(\texttt{False})
[-2, 1, 2, 0]
```

\textbf{Note:} The divisor \(D\) is \textit{q-reduced} if \(D = c + kq\) where \(c\) is superstable, \(k\) is an integer, and \(q\) is the sink.

```python
rank(\texttt{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:

```sage
sage: S = sandpiles.Complete(4)
sage: D = SandpileDivisor(S,[4,2,0,0])
sage: D.rank()
```

(continues on next page)
sage: D.rank(True)
(3, {0: 3, 1: 0, 2: 1, 3: 0})
sage: E = _[1]
sage: (D - E).rank()
-1

Riemann-Roch theorem:

sage: D.rank() - (S.canonical_divisor()-D).rank() == D.deg() + 1 - S.genus()
True

Riemann-Roch theorem:

sage: D.rank() - (S.canonical_divisor()-D).rank() == D.deg() + 1 - S.genus()
True
sage: S = Sandpile({0:[1,2],1:[0,-2,0,3],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}]
sage: D.rank(True)
(0, {0: 0, 1: 0, 2: 1})
sage: E = D.rank(True)[1]
sage: (D - E).effective_div()
[]

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 $\text{deg}(E) = r$.

\textbf{sandpile()}

The divisor’s underlying sandpile.

\textbf{OUTPUT:}

Sandpile

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}
**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:

```python
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:

```python
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:

```python
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:
```
sage: S = sandpiles.Cycle(4)
sage: D = SandpileDivisor(S, [0,0,1,1])
sage: D.support()
[2, 3]
sage: S.vertices(sort=True)
[0, 1, 2, 3]
```

unstable()
The unstable vertices.

OUTPUT:
list of vertices

EXAMPLES:
```
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:
```
sage: S = Sandpile({'a': ['c', 'b'], 'b': ['c', 'a'], 'c': ['a']}, 'a')
sage: D = SandpileDivisor(S, {'a':0, 'b':1, 'c':2})
sage: D.values()
[0, 1, 2]
sage: S.vertices(sort=True)
['a', 'b', 'c']
```

weierstrass_div(\texttt{\small verbose=True})
The Weierstrass divisor. Its value at a vertex is the weight of that vertex as a Weierstrass point. (See \texttt{\small SandpileDivisor.weierstrass_gap_seq}.)

INPUT:
\texttt{\small verbose} – (default: \texttt{\small True}) boolean

OUTPUT:
SandpileDivisor

EXAMPLES:

```python
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, 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}
```

`weierstrass_gap_seq(v='sink', weight=True)`

The Weierstrass gap sequence at the given vertex. If weight is True, then also compute the weight of each gap value.

INPUT:

- **v** – (default: sink) vertex
- **weight** – (default: True) boolean

OUTPUT:

list or (list of list) of integers

EXAMPLES:

```python
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(sort=True)]
[(1, 3), (1, 2), (1, 3), (1, 2)]
sage: D.weierstrass_gap_seq(v='sink')
# gap sequence at sink vertex, ∅
((1, 3), 1)
sage: D.weierstrass_rank_seq()
# rank sequence at the sink vertex
(1, 0, 0, -1)
```

**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 \( r, r - 1, \ldots, 0, -1, -1, \ldots \) and the rank sequence \( \text{rank}(D), \text{rank}(D - v), \text{rank}(D - 2v), \ldots \).

`weierstrass_pts(with_rank_seq=False)`

The Weierstrass points (vertices). Optionally, return the corresponding rank sequences.

INPUT:

- **with_rank_seq** – (default: False) boolean
OUTPUT:
tuple of vertices or list of (vertex, rank sequence)

EXAMPLES:

```python
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:

```python
sage: s = sandpiles.House()
sage: K = s.canonical_divisor()
sage: [K.weierstrass_rank_seq(v) for v in s.vertices(sort=True)]
[(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:

```python
sage: from sage.sandpiles.sandpile import admissible_partitions
sage: from sage.sandpiles.sandpile import partition_sandpile
sage: S = sandpiles.Cycle(4)
sage: P = [admissible_partitions(S, i) for i in [2,3,4]]
sage: P
[[[0, 2, 3], {1}],
 {0, 3}, {1, 2}],
 {0, 1, 3}, {2}],
```

(continues on next page)
(continued from previous page)

```python
sage: for p in P:
    ....: sum([partition_sandpile(S, i).betti(verbose=False)[-1] for i in p])
6
8
3
```

```python
sage: S.betti()
0 1 2 3
---
0: 1 - - -
1: - 6 8 3
---
total: 1 6 8 3
```

**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
sage: T = aztec_sandpile(2)
```

```python
sage: sorted(len(v) for u, v in T.items())
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 8]
```

```python
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
dictionary

'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(sort=True)]
[[], [[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
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: 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:

```python
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
```

(continues on next page)
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
                  -> superstables
    riemann-roch2 : directed graph with a superstable not majorized by a maximal
                  -> superstable
sage: S = sandlib('gor')
sage: S.resolution()
'R^1 <-- R^5 <-- R^5 <-- R^1'
```

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)
[ 3279 -79 -1599 -1600]
[ -1 1539 -136 -1402]
[ 0 -1 1650 -1649]
[ 0 0 -1658 1658]
``` 

REFERENCES:

• [PPW2013]

See also:

• sage.combinat.e_one_star
• sage.combinat.constellation
5.1 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 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: 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:
Discrete dynamics, Release 10.1

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'>

sage: P1.<x,y> = ProjectiveSpace(CC,1)
sage: H = End(P1)
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
        (y : x)
DynamicalSystem defaults to projective:

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)

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)

Note that domain is ignored if an endomorphism is passed in:

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

Constructing a common parent:

sage: P.<x,y> = ProjectiveSpace(ZZ, 1)
sage: DynamicalSystem([CC.0*x^2, 4/5*y^2])

(continues on next page)
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
(1.000000000000000*I*x^2 : 0.8000000000000000*y^2)

sage: P.<x,y> = ProjectiveSpace(GF(5), 1)
sage: K.<t> = GF(25)
sage: 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])
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 DynamicalSystem_projective that is this map coerced to $R$.

EXAMPLES:

```sage
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)
```

field_of_definition_critical($\text{return\_embedding}=False, \text{simplify\_all}=False, \text{names}='a'$)

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:

• $\text{return\_embedding}$ – (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.

• $\text{simplify\_all}$ – (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

• $\text{names}$ – (optional) string to be used as generator for returned number field or finite field

OUTPUT:

If $\text{return\_embedding}$ is False, the field of definition as an absolute number field or finite field. If $\text{return\_embedding}$ 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:

```sage
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)
```
Discrete dynamics, Release 10.1

\begin{verbatim}
 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

 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)
\end{verbatim}

\textbf{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:

- \texttt{n} – a positive integer
- \texttt{formal} – (default: \texttt{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. \texttt{False} specifies to find number field or finite field over which all periodic points of the n-th iterate are defined
- \texttt{return_embedding} – (default: \texttt{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: \texttt{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 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:

\begin{verbatim}
 sage: P.<x,y> = ProjectiveSpace(QQ, 1)
 sage: f = DynamicalSystem([x^2, y^2], domain=P)
\end{verbatim}

(continues on next page)
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: sorted(f.periodic_points(3, minimal=False, R=N), key=str)
[(-a^5 - a^4 - a^3 - a^2 - a - 1 : 1),
 (0 : 1),
 (1 : 0),
 (1 : 1),
 (a : 1),
 (a^2 : 1),
 (a^3 : 1),
 (a^4 : 1),
 (a^5 : 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:

```python
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 + 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`
• $\text{homset}$ – (optional) homset of specialized map

OUTPUT: `DynamicalSystem`

EXAMPLES:

```python
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)
```

5.1. Generic dynamical systems on schemes 115
5.2 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

```python
class sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine(polys_or_rat_fncts, domain)
```

Bases: `SchemeMorphism_polynomial_affine_space`, `DynamicalSystem`

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, 1])
Dynamical System of Affine Space of dimension 3 over Rational Field
    Defn: Defined on coordinates by sending (x, y, z) to
          (x, y, 1)
```
Discrete dynamics, Release 10.1

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
  Defn: Defined on coordinates by sending (x, y) to
        (9/4*x^2, 3/2*y)

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: H = End(A)
sage: f = H([x^2, y^2])
sage: DynamicalSystem_affine(f)
Dynamical System of Affine Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x, y) to
        (x^2, y^2)

Notice that \mathbb{Z}Z becomes \mathbb{Q}Q since the function is rational:

sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: DynamicalSystem_affine([3*x^2/(5*y), y^2/(2*x^2)])
Dynamical System of Closed subscheme of Affine Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x, y) to
        (3*x^2/(5*y), y^2/(2*x^2))

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: DynamicalSystem_affine([3/2*x^2, y^2])
Dynamical System of Affine Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x, y) to
        (3/2*x^2, y^2)

If you pass in quotient ring elements, they are reduced:

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)
Dynamical System of Closed subscheme of Affine Space of dimension 3 over Rational Field defined by:
  x - y

5.2. Dynamical systems on affine schemes

(continues on next page)
Defn: Defined on coordinates by sending 
(x, y, z) to 
(y, y, 2\*y)

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)
Dynamical System of Closed subscheme of Affine Space of dimension 3
over Univariate Polynomial Ring in t over Rational Field defined by:
x^2 - y^2
Defn: Defined on coordinates by sending 
(x, y, z) to 
(x^2/(t*y), t*y^2, x*z)

sage: x = var('x')
sage: DynamicalSystem_affine(x^2+1)
Traceback (most recent call last):
...  
TypeError: symbolic ring cannot be the base ring

conjugate(M)

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:

An affine dynamical system

EXAMPLES:

sage: A.<t> = AffineSpace(QQ, 1)
sage: f = DynamicalSystem_affine([t^2+1])
sage: f.conjugate(matrix([[1,2], [0,1]]))
Dynamical System of Affine Space of dimension 1 over Rational Field
Defn: Defined on coordinates by sending (t) to 
(t^2 + 4*t + 3)

sage: A.<x,y> = AffineSpace(ZZ,2)
sage: f = DynamicalSystem_affine([x^3+y^3, y^2])
sage: f.conjugate(matrix([[1,2,3], [0,1,2], [0,0,1]]))
Dynamical System of Affine Space of dimension 2 over Integer Ring
Defn: Defined on coordinates by sending (x, y) to 
(x^3 + 6*x^2*y + 12*x*y^2 + 9*y^3 + 9*x^2 + 36*x*y + 40*y^2 + 27*x + ~58*y + 28, y^2 + 4*y + 2)

sage: R.<x> = PolynomialRing(QQ)
sage: K.<i> = NumberField(x^2+1)

(continues on next page)
sage: A.<x> = AffineSpace(ZZ,1)
sage: f = DynamicalSystem_affine([x^3+2*x^2+3])
sage: f.conjugate(matrix([[i,i], [0,-i]]))
Dynamical System of Affine Space of dimension 1 over Integer Ring
  Defn: Defined on coordinates by sending (x) to
            (x^3 + x^2 - x - 5)

defree()
Return the degree of the affine dynamical system.

EXAMPLES:

sage: R.<c> = QuadraticField(7)
sage: A.<x,y,z> = AffineSpace(R, 3)
sage: f = DynamicalSystem_affine([x^2 + y^5 + c, x^11, z^19])
sage: f.degree()
19

sage: R.<c> = QQ[]
sage: A.<x> = AffineSpace(R, 1)
sage: f = DynamicalSystem_affine([x^4])
sage: f.degree()
4

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: f = DynamicalSystem_affine([x, y/(1 + x^2)])
sage: f.degree()
2

dynatomic_polynomial(period)
Compute the (affine) dynatomic polynomial of a dynamical system $f : A^1 \to A^1$.

The dynatomic polynomial is the analog of the cyclotomic polynomial and its roots are the points of formal period $n$.

ALGORITHM:
Homogenize to a map $f : P^1 \to P^1$ 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)
2^12*x^12 + 18^10*x^10 + 57^8*x^8 + 79^6*x^6 + 48^4*x^4 + 12^2*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 + 117.000000000000*x^4 + 78.0000000000000*x^2 + 1.00000000000000

sage: A.<x> = AffineSpace(QQ, 1)
\[ f = \text{DynamicalSystem_affine([x^2-10/9])} \]
sage: f.dynatomic_polynomial([2, 1])
531441*x^4 - 649539*x^2 - 524880

sage: A.<x> = AffineSpace(CC, 1)
\[ f = \text{DynamicalSystem_affine([x^2+CC.0])} \]
sage: f.dynatomic_polynomial(2)
x^2 + x + 1.00000000000000I + 1.00000000000000I

sage: K.<c> = FunctionField(QQ)
\[ A.<x> = AffineSpace(K, 1) \]
sage: f = DynamicalSystem_affine([x^2 + c])
sage: f.dynatomic_polynomial([1, 1])
z^2 + z + c

sage: R.<c> = QQ[]
\[ A.<z> = AffineSpace(R, 1) \]
sage: f = DynamicalSystem_affine([z^2 + c])
sage: f.dynatomic_polynomial([1, 1])
z^2 + c

sage: A.<x> = AffineSpace(CC, 1)
\[ F = \text{DynamicalSystem_affine([1/2*x^2 + CC(sqrt(3))])} \]
sage: F.dynatomic_polynomial([1, 1])
(0.125000000000000*x^4 + 0.366025403784439*x^2 + 1.5000000000000000)/(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: \texttt{DynamicalSystem\_projective}

EXAMPLES:

```sage
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
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 + (-2.00000000000000)*x2^4 : x0*x1^3 - x0^2*x1*x2)
```

```sage
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
sage: A.<x> = AffineSpace(QQ, 1)
sage: f = DynamicalSystem\_affine([x^2-1])
sage: f.homogenize((1, 0))
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)
```

```sage
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)
```

5.2. Dynamical systems on affine schemes 121
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 $\text{self.ccodomain().dimension_relative()}$ in the $\text{base_ring}$ of the map.

EXAMPLES:

```
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]
```

```
sage: P.<x,y,z> = AffineSpace(QQ, 3)
sage: f = DynamicalSystem_affine([x, y^2, z^2 - y])
sage: f.multiplier(P([1/2, 1, 0]), 2)
[1 0 0]
[0 4 0]
[0 0 0]
```

```
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]
```

```
sage: R.<t> = PolynomialRing(CC, 1)
sage: P.<x> = AffineSpace(R, 1)
sage: f = DynamicalSystem_affine([x^2 - t^2 + t])
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]
```
INPUT:

- P – a point in the map’s domain
- n – a positive integer

OUTPUT: a point in the map’s codomain

EXAMPLES:

```python
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, -9566677)
```

```python
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)
```

```python
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)
```

```python
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.
Discrete dynamics, Release 10.1

```
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))
```

```
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))
```

```
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)
```

```
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)]
```
Discrete dynamics, Release 10.1

```python
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)]
```

```python
class sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine_field(polys_or_rat_fncts, domain)

Bases: DynamicalSystem_affine, SchemeMorphism_polynomial_affine_space_field

reduce_base_field()

Return this map defined over the field of definition of the coefficients.

The base field of the map could be strictly larger than the field where all of the coefficients are defined. This function reduces the base field to the minimal possible. This can be done when the base ring is a number field, QQbar, a finite field, or algebraic closure of a finite field.

OUTPUT: A dynamical system

EXAMPLES:

```python
sage: K.<t> = GF(5^2)
sage: A.<x,y> = AffineSpace(K, 2)
sage: f = DynamicalSystem_affine([x^2 + 3*y^2, 3*y^2])
sage: f.reduce_base_field()
Dynamical System of Affine Space of dimension 2 over Finite Field of size 5
Defn: Defined on coordinates by sending (x, y) to
(x^2 - 2*y^2, -2*y^2)
```

```python
sage: A.<x,y> = AffineSpace(QQbar, 2)
sage: f = DynamicalSystem_affine([x^2 + QQbar(sqrt(3))*y^2, QQbar(sqrt(-1))*y^2])
sage: f.reduce_base_field()
Dynamical System of Affine Space of dimension 2 over Number Field in a with defining polynomial y^4 - y^2 + 1 with a = -0.866025403784439? + 0.
Defn: Defined on coordinates by sending (x, y) to
(x^2 + (a^3 - 2*a)*y^2, (a^3)*y^2)
```
```

class sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine_field

Bases: DynamicalSystem_affine, SchemeMorphism_polynomial_affine_space_field

reduce_base_field()

Return this map defined over the field of definition of the coefficients.

The base field of the map could be strictly larger than the field where all of the coefficients are defined. This function reduces the base field to the minimal possible. This can be done when the base ring is a number field, QQbar, a finite field, or algebraic closure of a finite field.

OUTPUT: A dynamical system

EXAMPLES:

```python
sage: K.<t> = GF(5^2)
sage: A.<x,y> = AffineSpace(K, 2)
sage: f = DynamicalSystem_affine([x^2 + 3*y^2, 3*y^2])
sage: f.reduce_base_field()
Dynamical System of Affine Space of dimension 2 over Finite Field of size 5
Defn: Defined on coordinates by sending (x, y) to
(x^2 - 2*y^2, -2*y^2)
```

```python
sage: A.<x,y> = AffineSpace(QQbar, 2)
sage: f = DynamicalSystem_affine([x^2 + QQbar(sqrt(3))*y^2, QQbar(sqrt(-1))*y^2])
sage: f.reduce_base_field()
Dynamical System of Affine Space of dimension 2 over Number Field in a with defining polynomial y^4 - y^2 + 1 with a = -0.866025403784439? + 0.
Defn: Defined on coordinates by sending (x, y) to
(x^2 + (a^3 - 2*a)*y^2, (a^3)*y^2)
```
```

class sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine_field

Bases: DynamicalSystem_affine, SchemeMorphism_polynomial_affine_space_field

reduce_base_field()

Return this map defined over the field of definition of the coefficients.

The base field of the map could be strictly larger than the field where all of the coefficients are defined. This function reduces the base field to the minimal possible. This can be done when the base ring is a number field, QQbar, a finite field, or algebraic closure of a finite field.

OUTPUT: A dynamical system

EXAMPLES:

```python
sage: K.<t> = GF(5^2)
sage: A.<x,y> = AffineSpace(K, 2)
sage: f = DynamicalSystem_affine([x^2 + 3*y^2, 3*y^2])
sage: f.reduce_base_field()
Dynamical System of Affine Space of dimension 2 over Finite Field of size 5
Defn: Defined on coordinates by sending (x, y) to
(x^2 - 2*y^2, -2*y^2)
```

```python
sage: A.<x,y> = AffineSpace(QQbar, 2)
sage: f = DynamicalSystem_affine([x^2 + QQbar(sqrt(3))*y^2, QQbar(sqrt(-1))*y^2])
sage: f.reduce_base_field()
Dynamical System of Affine Space of dimension 2 over Number Field in a with defining polynomial y^4 - y^2 + 1 with a = -0.866025403784439? + 0.
Defn: Defined on coordinates by sending (x, y) to
(x^2 + (a^3 - 2*a)*y^2, (a^3)*y^2)
```
In the cyclotomic field of order 5, define an affine space and a dynamical system:

```sage
K.<v> = CyclotomicField(5)
sage: A.<x,y> = AffineSpace(K, 2)
sage: f = DynamicalSystem_affine([(3*x^2 + y) / (5*x), (y^2+1) / (x+y)])
sage: f.reduce_base_field()
Dynamical System of Affine Space of dimension 2 over Rational Field
  Defined on coordinates by sending (x, y) to
  ((3*x^2 + y)/(5*x), (5*y^2 + 5)/(5*x + 5*y))
```

The `weil_restriction()` method computes the Weil restriction of this morphism over some extension field. If the field is a finite field, it computes the Weil restriction to the prime subfield. The Weil restriction is a functor that reduces questions about varieties over large fields to questions about more complicated varieties over smaller fields. This is useful for morphisms applied to a morphism giving the equivalent morphism from the Weil restriction of the domain to the Weil restriction of the codomain.

```sage
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 \( Res_{L/k} \) - is a functor which, for any finite extension of fields \( L/k \) and any algebraic variety \( X \) over \( L \), produces another corresponding variety \( Res_{L/k}(X) \), 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.
```

```sage
OUTPUT:
Scheme morphism on the Weil restrictions of the domain and codomain of the map.
EXAMPLES:

```sage
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
  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, 2*z2*z3)
```

```sage
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.
```

```sage
class sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine_finite_field(polys_or_rat_fncts, domain)

Bases: DynamicalSystem_affine_field, 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:

```
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
```

```
sage: A.<x,y> = AffineSpace(GF(13), 2)
sage: f = DynamicalSystem_affine([x^2 - 1, y^2])
sage: f.orbit_structure(A(2, 3))
[1, 6]
```

```
sage: A.<x,y,z> = AffineSpace(GF(49, 't'), 3)
sage: f = DynamicalSystem_affine([x^2 - z, x - y + z, y^2 - x^2])
sage: f.orbit_structure(A(1, 1, 2))
[7, 6]
```

5.3 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: SchemeMorphism_polynomial_projective_space, 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
Discrete dynamics, Release 10.1

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 $(X^2 - 3Y^2 : Y^2)$
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.<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 Univariate Polynomial Ring in t over Complex Field defined by:
  x - y
  Defn: Defined on coordinates by sending $(u : v : w)$ to $(u^2 : v^2 : w*u)$
```
2 over Complex Field with 53 bits of precision defined by:
\[ x - y \]
Defn: Defined on coordinates by sending \((x : y : z)\) to 
\((y^2 : y^2 : y^2z)\)

We can also compute the forward image of subschemes through elimination. In particular, let \(X = V(h_1, \ldots, h_t)\) and define the ideal \(I = (h_1, \ldots, h_t, y_0 - f_0(\bar{x}), \ldots, y_n - f_n(\bar{x}))\). Then the elimination ideal \(I_{n+1} = I \cap K[y_0, \ldots, y_n]\) 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 x P^1 over Rational Field
Defn: Defined by sending \((x : y : z , w : u)\) to 
\((x^2*u : y^2*w : z^2*u , w^2 : u^2).\)
```

```
sage: K.<v> = QuadraticField(-7)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: f = DynamicalSystem([(x^3 + v*x*y^2, y^3)])
sage: fbar = f.change_ring(QQbar)
sage: fbar.is_postcritically_finite()
False
```

**affine_preperiodic_model** \((m, n, return_conjugation=False)\)

Return a dynamical system conjugate to this one with affine \((n, m)\) preperiodic points.

If the base ring of this dynamical system is finite, there may not be a model with affine preperiodic points, in which case a ValueError is thrown.

**INPUT:**
- \(m\) – the preperiod of the preperiodic points to make affine.
- \(n\) – the period of the preperiodic points to make affine.
- **return_conjugation** – (default: False) If True, return a tuple \((g, \phi)\) where \(g\) is a model with affine \((n, m)\) preperiodic points and \(\phi\) is the matrix that moves \(f\) to \(g\).

**OUTPUT:** a dynamical system conjugate to this one.
EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([x^2, y^2, z^2])
sage: g = f.affine_preperiodic_model(0, 1); g
Dynamical System of Projective Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x : y : z) to
  (-x^2 : -2*x^2 + 2*x*y - y^2 : 2*x^2 - 2*x*y + 2*y^2 + 2*y*z + z^2)
```

We can check that \( g \) has affine fixed points:

```
sage: g.periodic_points(1)
[(-1 : -1 : 1),
 (-1/2 : -1 : 1),
 (-1/2 : -1/2 : 1),
 (-1/3 : -2/3 : 1),
 (0 : -1 : 1),
 (0 : -1/2 : 1),
 (0 : 0 : 1)]
```

```
sage: P.<x,y,z> = ProjectiveSpace(GF(9), 2)
sage: f = DynamicalSystem_projective([x^2, y^2, z^2])
sage: f.affine_preperiodic_model(0, 1)
Dynamical System of Projective Space of dimension 2 over Finite Field in z2 of size 3^2
  Defn: Defined on coordinates by sending (x : y : z) to
  (-z2)*x^2 : z2*x^2 + (-z2)*x*y + (-z2)*y^2 :
  (-z2)*x^2 + z2*x*y + (z2 + 1)*y^2 - y*z + z^2)
```

```
sage: R.<c> = GF(3)[]
sage: P.<x,y,z> = ProjectiveSpace(R, 2)
sage: f = DynamicalSystem_projective([x^2, y^2, z^2])
sage: f.affine_preperiodic_model(0, 1)  # long time
Dynamical System of Projective Space of dimension 2 over Finite Field in c over Finite Field of size 3
  Defn: Defined on coordinates by sending (x : y : z) to
  (2*c^3*x^2 : c^3*x^2 + 2*c^3*x*y + 2*c^3*y^2 :
  2*c^3*x^2 + c^3*x*y + (c^3 + c^2)*y^2 + 2*c^2*y*z + c^2*z^2)
```

```
sage: K.<k> = CyclotomicField(3)
sage: P.<x,y,z> = ProjectiveSpace(K, 2)
sage: f = DynamicalSystem_projective([x^2 + k*x*y + y^2, z^2, y^2])
sage: f.affine_preperiodic_model(1, 1)
Dynamical System of Projective Space of dimension 2 over Cyclotomic Field of order 3 and degree 2
  Defn: Defined on coordinates by sending (x : y : z) to
  (-y^2 : x^2 : x^2 + (-k)*x*z + z^2)
```

```
sage: K.<k> = CyclotomicField(3)
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2 + y^2, y^2])
sage: g, mat = f.affine_preperiodic_model(0, 1, return_conjugation=True)
sage: g == f.conjugate(mat)
True
```

5.3. Dynamical systems on projective schemes 131
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: X = P.subscheme(2*y - z)
sage: f = DynamicalSystem_projective([x^2 + y^2, z^2 + y^2, z^2], domain=X)
sage: f.affine_preperiodic_model(0, 1)
Dynamical System of Closed subscheme of Projective Space of dimension 2
over Rational Field defined by: 2*y - z
Defn: Defined on coordinates by sending (x : y : z) to
(-x^2 - y^2 : y^2 : x^2 + z^2)

all_minimal_models(return_transformation=False, prime_list=None, algorithm=None,
check_minimal=True)
Determine a representative in each $SL(2, \mathbb{Z})$-orbit of this map.
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 $PGL_2$ transforma-
tion 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:

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)]

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: c = 2^3*3^6
sage: f = DynamicalSystem([x^3 - c^2*y^3, x*y^2])
sage: len(f.all_minimal_models(algorithm='HS'))
14

(continues on next page)
sage: len(f.all_minimal_models(prime_list=[2], algorithm='HS'))
2

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem([237568*x^3 + 1204224*x^2*y + 2032560*x*y^2
....: + 1142289*y^3, -131072*x^3 - 663552*x^2*y - 1118464*x*y^2
....: - 627664*y^3])
sage: len(f.all_minimal_models(algorithm='BM'))
2

REFERENCES:

• [BM2012]
• [HS2018]

arakelov_zhang_pairing(g, **kwds)

Return an estimate of the Arakelov-Zhang pairing of the rational maps self and g on $\mathbb{P}^1$ over a number field.

The Arakelov-Zhang pairing was introduced by Petsche, Szpiro, and Tucker in 2012, which measures the dynamical closeness of two rational maps. They prove inter alia that if one takes a sequence of small points for one map (for example, preperiodic points for self) and measure their dynamical height with respect to the other map (say, g), then the values of the height will tend to the value of the Arakelov-Zhang pairing.

The Arakelov-Zhang pairing involves mutual energy integrals between dynamical measures, which are in the case of polynomials, the equilibrium measures of the associated Julia sets at each place. As a result, these pairings are very difficult to compute exactly via analytic methods. We use a discrete approximation to these energy integrals.

ALGORITHM:

We select periodic points of order n, or n-th preimages of a specified starting value given by f_starting_point and g_starting_point. At the archimedean places and the places of bad reduction of the two maps, we compute the discrete approximations to the energy integrals involved using these points.

INPUT:

• g - a rational map of $\mathbb{P}^1$ given as a projective morphism. g and self should have the same field of definition.

kwds:

• n - (default: 5) a positive integer Order of periodic points to use or preimages to take if starting points are specified.

• f_starting_point - (optional, default: None) value in the base number field or None. If f_starting_point is None, we solve for points of period n for self. Otherwise, we take n-th preimages of the point given by f_starting_point under f on the affine line.

• g_starting_point - (optional, default: None) value in the base number field or None. If g_starting_point is None, we solve for points of period n for g. Otherwise, we take n-th preimages of the point given by g_starting_point under g on the affine line.

• check_primes_of_bad_reduction - (optional, default: False) boolean. Passed to the primes_of_bad_reduction function for self and g.

• prec - (optional, default: RealField default) default precision for RealField values which are returned.
noise_multiplier - (default: 2) a real number. Discriminant terms involved in the computation at the archimedean places are often not needed, particularly if the capacity of the Julia sets is 1, and introduce a lot of error. By a well-known result of Mahler (see also M. Baker, “A lower bound for averages of dynamical Green’s functions”) such error (for a set of \( N \) points) is on the order of \( \log(N)/N \) after our normalization. We check if the value of the archimedean discriminant terms is within \( 2 \times \text{noise}_\text{multiplier} \) of \( \log(N)/N \). If so, we discard it. In practice this greatly improves the accuracy of the estimate of the pairing. If desired, \text{noise}_\text{multiplier} \) can be set to 0, and no terms will be ignored.

**OUTPUT:**

- a real number estimating the Arakelov-Zhang pairing of the two rational maps.

**EXAMPLES:**

```sage
sage: K.<k> = CyclotomicField(3)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: f = DynamicalSystem_projective([x^2 + (2*k + 2)*y^2, y^2])
sage: g = DynamicalSystem_projective([x^2, y^2])
sage: pairingval = f.arakelov_zhang_pairing(g, n=5); pairingval
0.409598197761958
```

```sage
definitions:
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2 + 4*y^2, y^2])
sage: g = DynamicalSystem_projective([x^2, y^2])
sage: pairingval = f.arakelov_zhang_pairing(g, n=6); pairingval
0.750178391443644
```

```sage
# Compare to the exact value:
sage: dynheight = f.canonical_height(P(0, 1)); dynheight
0.75017839144364417318023000563
```

```sage
sage: dynheight - pairingval
0.000000000000000
```

Notice that if we set the noise_multipler to 0, the accuracy is diminished:

```sage
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2 + 4*y^2, y^2])
sage: g = DynamicalSystem_projective([x^2, y^2])
sage: pairingval = f.arakelov_zhang_pairing(g, n=6, noise_multiplier=0)
sage: pairingval
0.650660018921632
```

```sage
sage: dyheight = f.canonical_height(P(0, 1)); dynheight
0.75017839144364417318023000563
```

```sage
sage: dynheight - pairingval
-0.0995183725220122
```

We compute the example of Prop. 18(d) from Petsche, Szpiro and Tucker:

```sage
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([y^2 - (y - x)^2, y^2])
sage: g = DynamicalSystem_projective([x^2, y^2])
sage: f.arakelov_zhang_pairing(g)
0.326954667248466
```

```sage
# Correct value should be = 0.323067...
```

```sage
# long time
```

(continues on next page)
0.323091061918965
sage: _ - 0.323067            # long time
0.0000240619189654789

Also from Prop. 18 of Petsche, Szpiro and Tucker, includes places of bad reduction:

sage: R.<z> = PolynomialRing(ZZ)
sage: K.<b> = NumberField(z^3 - 11)
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: a = 7/(b - 1)
sage: f = DynamicalSystem_projective([a*y^2 - (a*y - x)^2, y^2])
sage: g = DynamicalSystem_projective([x^2, y^2])

If all archimedean absolute values of a have modulus > 2, then the pairing should be h(a):

sage: f.arakelov_zhang_pairing(g, n=6)  # long time
1.93846423207664
sage: _ - a.global_height()               # long time
-0.00744591697867292

automorphism_group(**kwds)

Calculates the subgroup of PGL2 that is the automorphism group of this dynamical system.

The automorphism group is the set of PGL(2) elements that fixes this map under conjugation.

INPUT:

The following keywords are used in most cases:

- `num_cpus` – (default: 2) the number of threads to use. Setting to a larger number can greatly speed up this function.

The following keywords are used only when the dimension of the domain is 1 and the base ring is the rationals, but ignored in all other cases:

- `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:

5.3. Dynamical systems on projective schemes 135
Discrete dynamics, Release 10.1

```
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: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem([x^3, y^3, z^3])
sage: len(f.automorphism_group())
24

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: lst, label = f.automorphism_group(algorithm='CRT', return_functions=True, iso_type=True)
sage: sorted(lst), label
([-1/x, 1/x, (-x - 1)/(x - 1), (-x + 1)/(x + 1), (x - 1)/(x + 1),
  (x + 1)/(x - 1), -x, x],
 'Dihedral of order 8')

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()
[ [1 0 0]
[0 1 0]
[0 0 1]]
```

Chapter 5. Arithmetic Dynamical Systems
Discrete dynamics, Release 10.1

\begin{Verbatim}
sage: K.<w> = CyclotomicField(3)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: D6 = DynamicalSystem_projective([y^2, x^2])
sage: sorted(D6.automorphism_group())
\[
[[-w - 1 0], [0 -w - 1], [w 0], [0 w], [0 1], [1 0],
[ 0 1], [1 0], [0 1]
]
\end{Verbatim}

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 to not 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:**

\begin{Verbatim}
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
sage: f.canonical_height(P.point([2,1]), error_bound=0.001)
1.0984430632822307984974382955
\end{Verbatim}

Notice that preperiodic points may not return exactly 0:

\begin{Verbatim}
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
5.73649197870160119266380480e-8
sage: f.nth_iterate(Q, 2) == Q # but it is indeed preperiodic
True
\end{Verbatim}

5.3. Dynamical systems on projective schemes

137
**conjugate***(M, adjugate=False, normalize=False)***

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
- adjugate – (default: False) boolean, also classically called adjoint, takes a square matrix \( M \) and finds the transpose of its cofactor matrix. Used for conjugation in place of inverse when specified True. Functionality is the same in projective space.
• normalize – (default: False) boolean, if normalize is True, then the method normalize_coordinates is called.

OUTPUT: a dynamical system

EXAMPLES:

```sage
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)
```

```sage
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)
```

```sage
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)
```

```sage
sage: P.<x,y> = ProjectiveSpace(ZZ,1)
sage: f = DynamicalSystem_projective([x^2+y^2, y^2])
sage: f.conjugate(matrix([[2,0], [0,1/2]]))
Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to
  (2*x^2 + 1/8*y^2 : 1/2*y^2)
```

```sage
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)
```

Todo: Use the left and right action functionality to replace the code below with #return DynamicalSystem_projective(M.inverse()*self*M, domain=self.codomain()) once there is a function to pass to the smallest field of definition.
critical_height(**kwds)

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.

The computations can be done either over the algebraic closure of the base field or over the minimal extension of the base field that contains the critical points.

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
• use_algebraic_closure – boolean (default: True) – If True, uses the algebraic closure. If False, uses the smallest extension of the base field containing all the critical points.

OUTPUT: real number

EXAMPLES:

\begin{verbatim}
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: 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
\end{verbatim}

Postcritically finite maps have critical height 0:

\begin{verbatim}
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)
0.000000000000000000000000000000

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^3 + 3*x*y^2, y^3])
sage: f.critical_height(use_algebraic_closure=False)
0.000023477016733897112886491967991
sage: f.critical_height()
0.000023477016733897112886491967991
\end{verbatim}

critical_point_portrait(check=True, use_algebraic_closure=True)

If this dynamical system is post-critically finite, return its critical point portrait.

This is the directed graph of iterates starting with the critical points. Must be dimension 1. If check is True, then the map is first checked to see if it is postcritically finite.
The computations can be done either over the algebraic closure of the base field or over the minimal extension of the base field that contains the critical points.

**INPUT:**

- check – boolean (default: True)
- use_algebraic_closure – boolean (default: True) – If True, uses the algebraic closure. If False, uses the smallest extension of the base field containing all the critical points.

**OUTPUT:** a digraph

**EXAMPLES:**

```sage
R.<z> = QQ[]
K.<v> = NumberField(z^6 + 2*z^5 + 2*z^4 + 2*z^3 + z^2 + 1)
P.<x,y> = ProjectiveSpace(K,1)
f = DynamicalSystem_projective([x^2 + v*y^2, y^2])
f.critical_point_portrait(check=False)  # long time
Looped digraph on 6 vertices
```

```sage
P.<x,y> = ProjectiveSpace(K,1)
f = DynamicalSystem_projective([x^5 + 5/4*x*y^4, y^5])
f.critical_point_portrait(check=False)
Looped digraph on 5 vertices
```

```sage
P.<x,y> = ProjectiveSpace(QQ,1)
f = DynamicalSystem_projective([x^2 + 2*y^2, y^2])
f.critical_point_portrait()
Traceback (most recent call last):
...  
TypeError: map must be post-critically finite
```

```sage
R.<t> = QQ[]
K.<v> = NumberField(t^3 + 2*t^2 + t + 1)
phi = K.embeddings(QQbar)[0]
P.<x, y> = ProjectiveSpace(K, 1)
f = DynamicalSystem_projective([x^2 + v*y^2, y^2])
f.change_ring(phi).critical_point_portrait()
Looped digraph on 4 vertices
```

```sage
P.<x,y> = ProjectiveSpace(QQ,1)
f = DynamicalSystem_projective([8*x^4 - 8*x^2*y^2 + y^4, y^4])
f.critical_point_portrait(use_algebraic_closure=False)  # long time
Looped digraph on 6 vertices
```

```sage
P.<x,y> = ProjectiveSpace(QQbar,1)
f = DynamicalSystem_projective([8*x^4 - 8*x^2*y^2 + y^4, y^4])
f.critical_point_portrait()  # long time
Looped digraph on 6 vertices
```

```sage
P.<x,y> = ProjectiveSpace(GF(3),1)
f = DynamicalSystem_projective([x^2 + x*y - y^2, x*y])
f.critical_point_portrait(use_algebraic_closure=False)
```

(continues on next page)
Looped digraph on 6 vertices
`sage`: `f.critical_point_portrait()` #long time
Looped digraph on 6 vertices

**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:**

```
sage: set_verbose(None)
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^3 - 2*x^2*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:**

```
sage: set_verbose(None)
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^3 - 2*x^2*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
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_subscheme()
Closed subscheme of Projective Space of dimension 1 over Rational Field defined by:
4*x^2 + 2*y^2
```
Discrete dynamics, Release 10.1

```python
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
```

```python
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
```

```python
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:**

```python
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]
```

```python
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]
```

```python
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]
```

```python
sage: P2.<X,Y,Z> = ProjectiveSpace(ZZ, 2)
sage: f = DynamicalSystem_projective([X^2, Y^2, Z^2])
sage: f.degree_sequence(10)
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

**dehomogenize**(*n*)

Return the standard dehomogenization at the n[0] coordinate for the domain and the n[1] coordinate for the codomain.

5.3. Dynamical systems on projective schemes 143
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 \(\lim_{n \to \infty} \sqrt[n]{\text{deg}(f^n)}\).

**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)
1.43096908110525501452244131
```

**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:**

```sage
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: 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

sage: P.<x,y> = ProjectiveSpace(CC,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
```

5.3. Dynamical systems on projective schemes 145
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))
(continues on next page)
\begin{verbatim}
sage: ff(ff(p)) == p
  True

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
   + (0.1111111111111111 - 0.3333333333333333*I)*y^5)/(x^2*y - x*y^2
   + (0.3333333333333333*I)*y^3)

sage: P.<x,y> = ProjectiveSpace(ZZ, 1)
sage: f = DynamicalSystem_projective([x^2 - 5*y^2, y^2])
sage: f.dynatomic_polynomial([3, 0])
  0

\end{verbatim}

\textbf{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 \( \mathbb{Q} \) or field embedding. Non-archimedean places are prime ideals for number fields or primes over \( \mathbb{Q} \).

\textbf{ALGORITHM:}

See Exercise 5.29 and Figure 5.6 of [Sil2007].

\textbf{INPUT:}
Discrete dynamics, Release 10.1

- $P$ – 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:

```sage
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
```

```sage
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
```

```sage
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
```

```sage
sage: f.green_function(P([2, 1]), K.ideal(7), N=7)
0.4864775372682832627633818586
```

```sage
sage: f.green_function(P([w, 1]), K.ideal(17), error_bound=0.001)
-0.70813041039490996737374178059
```

```sage
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 + y^2, x*y])
sage: f.green_function(P.point([5,2], False), 0, N=30)
1.731545184477407992085512000
```

```sage
sage: f.green_function(P.point([2,1], False), 0, N=30)
0.8657725922318108832526209926
```

```sage
sage: f.green_function(P.point([1,1], False), 0, N=30)
0.4328862961086233861270146098
```

$\text{height\_difference\_bound}(\text{prec=}\text{None})$

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:
• prec – (default: 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

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()
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()
11.3683039374269
```

```
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()
2.89037175789616
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem([5*x^2 + 3*x*y , y^2 + 3*x^2])
sage: f.height_difference_bound(prec=100)
5.33753807970131797372224159274
```

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])
(continues on next page)
```
Discrete dynamics, Release 10.1

\begin{verbatim}
(sage) sage: f.is_PGL_minimal()
False

(sage) sage: PS.<x,y> = ProjectiveSpace(QQ,1)
(sage) sage: f = DynamicalSystem_projective([6*x^2 + 12*x*y + 7*y^2, y^2])
(sage) sage: f.is_PGL_minimal()
False

is_dynamical_belyi_map()

Return if this dynamical system is a dynamical Belyi map.

We define a dynamical Belyi map to be a map conjugate to a dynamical system \( f : \mathbb{P}^1 \to \mathbb{P}^1 \) where the branch points are contained in \{0, 1, \infty\} and the postcritical set is contained in \{0, 1, \infty\}.

Output: Boolean

EXAMPLES:

(sage) sage: P.<x,y> = ProjectiveSpace(QQ, 1)
(sage) sage: f = DynamicalSystem_projective([-2*x^3 - 9*x^2*y - 12*x*y^2 - 6*y^3, y^3])
(sage) sage: f.is_dynamical_belyi_map()
True

(sage) sage: P.<x,y> = ProjectiveSpace(QQ, 1)
(sage) sage: f = DynamicalSystem_projective([5*x^7 - 7*x^6*y, -7*x*y^6 + 5*y^7])
(sage) sage: f.is_dynamical_belyi_map()
True

(sage) sage: P.<x,y> = ProjectiveSpace(QQ, 1)
(sage) sage: f = DynamicalSystem_projective([x^2 + y^2, y^2])
(sage) sage: f.is_dynamical_belyi_map()
False

(sage) sage: F = QuadraticField(-7)
(sage) sage: P.<x,y> = ProjectiveSpace(F, 1)
(sage) sage: f = DynamicalSystem_projective([5*x^7 - 7*x^6*y, -7*x*y^6 + 5*y^7])
(sage) sage: f.is_dynamical_belyi_map()
True

(sage) sage: P.<x,y> = ProjectiveSpace(QQ, 1)
(sage) sage: f = DynamicalSystem_projective([2*x^3 + 3*x^2*y - 3*x*y^2 + 2*y^3, x^3 + y^3])
(sage) sage: f.is_dynamical_belyi_map()
False

(sage) sage: R.<t> = PolynomialRing(QQ)
(sage) sage: N.<c> = NumberField(t^3 - 2)
(sage) sage: P.<x,y> = ProjectiveSpace(N, 1)
(sage) sage: f = DynamicalSystem_projective([x^2 + c*y^2, x*y])
(sage) sage: f.is_dynamical_belyi_map()
False
\end{verbatim}
sage: P.<x,y> = ProjectiveSpace(GF(7), 1)
sage: f = DynamicalSystem_projective([x^3 + 6*y^3, y^3])
sage: f.is_dynamical_belyi_map()
False

is_postcritically_finite(err=0.01, use_algebraic_closure=True)

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 err is passed into is_preperiodic() as part of the preperiodic check.

The computations can be done either over the algebraic closure of the base field or over the minimal extension of the base field that contains the critical points.

INPUT:

- err – (default: 0.01) positive real number
- use_algebraic_closure – boolean (default: True) – If True, uses the algebraic closure. If False, uses the smallest extension of the base field containing all the critical points.

OUTPUT: boolean

EXAMPLES:

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 - y^2, y^2])
sage: f.is_postcritically_finite()
True

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^3 - y^3, y^3])
sage: f.is_postcritically_finite()
False

sage: R.<z> = QQ[]
sage: K.<v> = NumberField(z^8 + 3*z^6 + 3*z^4 + z^2 + 1)
sage: FS.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x^3 + v*y^3, y^3])
sage: f.is_postcritically_finite() # long time
True

sage: K = UniversalCyclotomicField()
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: F = DynamicalSystem_projective([x^2 - y^2, y^2], domain=P)
sage: F.is_postcritically_finite()
True
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([8*x^4 - 8*x^2*y^2 + y^4, y^4])
sage: f.is_postcritically_finite(use_algebraic_closure=False) # long time
True

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^4 - x^2*y^2 + y^4, y^4])
sage: f.is_postcritically_finite(use_algebraic_closure=False)
False

sage: P.<x,y> = ProjectiveSpace(QQbar,1)
sage: f = DynamicalSystem_projective([x^4 - x^2*y^2, y^4])
sage: f.is_postcritically_finite()
False

minimal_model(return_transformation=False, prime_list=None, algorithm=None, check_primes=True)
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:

• return_transformation – (default: False) boolean; this signals a return of the $PGL_2$ transformation to conjugate this map to the calculated minimal model

• 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:

  • check_primes – (optional) boolean: this signals whether to check whether each element in prime_list is a prime

    – '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:

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
  Defined on coordinates by sending (X : Y) to
  (X^2 + 3*Y^2 : X*Y)
),
[1 0]

(continues on next page)
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),

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)

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

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem([2*x^2, y^2])
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 : 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)

REFERENCES:

• [BM2012]
• [Mol2015]
multiplier\((P, n, \text{check=True})\)

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 \(P\)
- \(\text{check}\) – (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]
```

```python
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]
```

```python
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]
```

```python
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]
```

```python
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)]
```

```python
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):
  ...
ValueError: (0 : 1) is not periodic of period 1
```
multiplier_spectra(n, formal=False, type='point', use_algebraic_closure=True, check=True)

Computes the \( n \) multiplier spectra of this dynamical system.

This is the set of multipliers of all periodic points of period \( n \) included with the appropriate multiplicity. User can also specify to compute the formal \( n \) multiplier spectra instead which includes the multipliers of all formal periodic points of period \( n \) with appropriate multiplicity. The map must be defined over projective space over a number field or finite field.

By default, the computations are done over the algebraic closure of the base field. If the map is defined over projective space of dimension 1, the computation can be done over the minimal extension of the base field that contains the periodic points. Otherwise, it will be done over the base ring of the map.

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
- type – (default: 'point') string; either 'point' or 'cycle' depending on whether you compute one multiplier per point or one per cycle
- use_algebraic_closure – boolean (default: True) – If True uses the algebraic closure. Using the algebraic closure can sometimes lead to numerical instability and extraneous errors. For most accurate results in dimension 1, set to False. If False, and the map is defined over projective space of dimension 1, uses the smallest extension of the base field containing all the periodic points. If the map is defined over projective space of dimension greater than 1, then the base ring of the map is used.
- check – (default: True) whether to check if the full multiplier spectra was computed. If False, can lead to mathematically incorrect answers in dimension greater than 1. Ignored if use_algebraic_closure is True or if this dynamical system is defined over projective space of dimension 1.

OUTPUT:

A list of field elements if the domain of the map is projective space of dimension 1. If the domain of the map is projective space of dimension greater than 1, a list of matrices

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2 - 3/4*y^2, y^2])
sage: sorted(f.multiplier_spectra(2, type='point'))
[0, 1, 1, 1, 9]
sage: sorted(f.multiplier_spectra(2, type='cycle'))
[0, 1, 1, 9]
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([x^2, z^2, y^2])
sage: f.multiplier_spectra(1)

[2 1 -1.732050807568878?I]
[0 0 -2],
[2 1 +1.732050807568878?I] [0 0] [0 0]
[0 0 -2], [0 -2], [0 -2],
[0 0] [0 0] [2 -2]
[0 -2], [0 0], [0 -2]
```
Discrete dynamics, Release 10.1

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([x^2, z^2, y^2])
sage: f.multiplier_spectra(2, formal=True)  # long time
[[4 0] [4 0] [4 0] [4 0] [4 0] [4 0] [4 0] [4 0] [0 0] [0 0] [0 0], [0 0] [0 0] [0 0] [0 0] [4 0] [4 0] [4 0] [0 0] [0 0] [0 0] [0 0], [4 0] [4 0] [4 0] [4 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] [0 0] [0 0] [0 0]]

sage: set_verbose(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: sorted(f.multiplier_spectra(2, formal=False, type='cycle'))
[0, 0.0681483474218635? - 1.930649271699173?*I, 0.0681483474218635? + 1.930649271699173?*I, 5.931851652781377? + 0.?e-49*I]

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([4608*x^10 - 2910096*x^9*y + 325988068*x^8 - 5*y^2
... - 3182519832*x^7*y^3 - 4139806626613*x^6*y^4 - 44439736715486*x^5*y^5
... + 2317935971590902*x^4*y^6 - 15344764859590852*x^3*y^7
... + 2561851642765275*x^2*y^8 + 113578270285012470*x*y^9
... - 150049940203963800*y^10, 4608*y^10])
sage: sorted(f.multiplier_spectra(1))
[-119820502365680843999, -7198147681176255644585/256, -3086380435599991/9, -32327381962860268721722583135/35184372088832, -429099949949944936653/2097152, 0, 529278480109921/256, 1061953534167447403/19683, 84844615755684859363/19683, 82911372672808161930567/8192, 3553497751559301575157261317/8192]

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]
```

(continues on next page)
\(\begin{array}{l}
[0, \\
-1, \\
1/128*a^5 - 13/384*a^4 + 5/96*a^3 + 1/16*a^2 + 43/128*a + 303/128, \\
-1/288*a^5 + 1/96*a^4 + 1/24*a^3 - 1/3*a^2 + 5/32*a - 115/32, \\
-5/1152*a^5 + 3/128*a^4 - 3/32*a^3 + 13/48*a^2 - 63/128*a - 227/128]
\end{array}\)

sage: f.multiplier_spectra(1)
\[0, -1, 1.951373035591442?, -2.475686517795721? - 0.730035681602057?*I, -2.475686517795721? + 0.730035681602057?*I]\n
sage: P.<x,y> = ProjectiveSpace(GF(5), 1)
sage: f = DynamicalSystem_projective([x^4 + 2*y^4, 4*x^2*y^2])
sage: f.multiplier_spectra(1)
\[0, 3*a + 3, 2*a + 1, 1, 1]\n
sage: f.multiplier_spectra(1)
\[0, 2*z2 + 1, 3*z2 + 3, 1, 1]\n
sage: K = GF(3).algebraic_closure()
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: f = DynamicalSystem_projective([x^5 + 2*y^5, 4*x^3*y^2])
sage: f.multiplier_spectra(1)
\[0, z3 + 2, z3 + 1, z3, 1, 1]\n
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?

INPUT:
- \(P\) – a point in this map’s domain
- \(n\) – a positive integer

kwds:
- normalize – (default: False) boolean

OUTPUT: a point in this map’s codomain
EXAMPLES:

```
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)
```

```
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)
```

```
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)
```

```
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)
```

```
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)
```

```
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)
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: f = DynamicalSystem_projective([x^2 + 3*y^2, 2*y^2, z^2])
sage: f.nth_iterate(P(2, 7, 1), -2)
Traceback (most recent call last):
  ...
TypeError: must be a forward orbit
```

```
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)
(continues on next page)
```
Traceback (most recent call last):
...
ValueError: [0, 0] does not define a point in Projective Space of dimension 1 over Rational Field since all entries are zero

```
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)
```

```
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem([x + y, y])
sage: Q = (3,1)
sage: f.nth_iterate(Q,0)
(3 : 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 + (-7.00000000000000e+00)*x^6*y^2 + 13.00000000000000e+00*x^4*y^4
         + (-7.00000000000000e+00)*x^2*y^6 + y^8
```

(continues on next page)
Discrete dynamics, Release 10.1

(continued from previous page)

\[ x^7 y + (-4.00000000000000) x^5 y^3 + 4.00000000000000 x^3 y^5 - x^7 \]

```python
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)
```

```python
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)
```

```python
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)
```

\texttt{nth\_preimage\_tree}(Q, n,**kwds)

Return the $n$-th pre-image tree rooted at $Q$.

This map must be an endomorphism of the projective line defined over a number field, algebraic field, or finite field.

**INPUT:**

- $Q$ – a point in the domain of this map
- $n$ – a positive integer, the depth of the pre-image tree

**kwds:**

- \texttt{return\_points} – (default: False) boolean; if True, return a list of lists where the index $i$ is the level of the tree and the elements of the list at that index are the $i$-th preimage points as an algebraic element of the splitting field of the polynomial $f^n - Q = 0$
- \texttt{numerical} – (default: False) boolean; calculate pre-images numerically. Note if this is set to True, preimage points are displayed as complex numbers
- \texttt{prec} – (default: 100) positive integer; the precision of the ComplexField if we compute the preimage points numerically
- \texttt{display\_labels} – (default: True) boolean; whether to display vertex labels. Since labels can be very cluttered, can set \texttt{display\_labels} to False and use \texttt{return\_points} to get a hold of the points themselves, either as algebraic or complex numbers

160 Chapter 5. Arithmetic Dynamical Systems
Discrete dynamics, Release 10.1

- **display_complex** – (default: False) boolean; display vertex labels as complex numbers. Note if this option is chosen that we must choose an embedding from the splitting field field_def of the \( n \)-th pre-image equation into \( \mathbb{C} \). We make the choice of the first embedding returned by field_def.embeddings(ComplexField())

- **digits** – a positive integer, the number of decimal digits to display for complex numbers. This only applies if display_complex is set to True

OUTPUT:

If return_points is False, a GraphPlot object representing the \( n \)-th pre-image tree. If return_points is True, a tuple \((GP, points)\), where \( GP \) is a GraphPlot object, and points is a list of lists as described above under return_points.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 + y^2, y^2])
sage: Q = P(0,1)
sage: f.nth_preimage_tree(Q, 2)
GraphPlot object for Digraph on 7 vertices
```

```
sage: P.<x,y> = ProjectiveSpace(GF(3), 1)
sage: f = DynamicalSystem_projective([x^2 + x*y + y^2, y^2])
sage: Q = P(0,1)
sage: f.nth_preimage_tree(Q, 2, return_points=True)
(GraphPlot object for Digraph on 4 vertices,
[[[0 : 1]], [[[1 : 1]], [[0 : 1], [2 : 1]]]])
```

**orbit** \((P, N, **kwds)\)

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 normalize=True. Perform the checks on point initialization if check=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:

- **check** – (default: True) boolean
- **normalize** – (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])
(continues on next page)
```

5.3. Dynamical systems on projective schemes 161
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])
sage: f.orbit(P.point([1,2], False), 4, check=False)
[(1 : 2), (5 : 4), (41 : 16), (1937 : 256), (3817505 : 65536)]

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: f.orbit(P(2, 1), [-1, 4])
Traceback (most recent call last):
  ...TypeError: orbit bounds must be non-negative

sage: f.orbit(P(2, 1), 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: f.orbit(P(0, 1), 3)
Traceback (most recent call last):
  ...TypeError: [0, 0] does not define a point in Projective Space of dimension 1 over Rational Field since all entries are zero

sage: f.orbit(P(0, 1), 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: f.orbit(P(2,1), 3, normalize=False)
[(2 : 1), (8 : 2), (512 : 32), (134217728 : 524288)]

(continues on next page)
\texttt{sage: f.orbit(P(2, 1), 3, normalize=True)}
\[(2 : 1), (4 : 1), (16 : 1), (256 : 1)]

\texttt{sage: P.<x,y,z> = ProjectiveSpace(QQ,2)}
\texttt{sage: f = DynamicalSystem_projective([x^2, y^2, x*z])}
\texttt{sage: f.orbit((2/3, 1/3), 3)}

\textbf{periodic\_points\,(n, minimal=True, formal=False, R=None, algorithm='variety', return\_scheme=False)}

Computes the periodic points of period $n$ of this dynamical system defined over the ring $R$ or the base ring of the map.

This can be done either by finding the rational points on the variety defining the points of period $n$, or, for finite fields, finding the cycle of appropriate length in the cyclegraph. For small cardinality fields, the cyclegraph algorithm is effective for any map and length cycle, but is slow when the cyclegraph is large. The variety algorithm is good for small period, degree, and dimension, but is slow as the defining equations of the variety get more complicated.

For rational maps, where there are potentially infinitely many periodic points of a given period, you must use the \texttt{return\_scheme} option. Note that this scheme will include the indeterminacy locus.

INPUT:

- \texttt{n} - a positive integer
- \texttt{minimal} – (default: \texttt{True}) boolean; \texttt{True} specifies to find only the periodic points of minimal period $n$ and \texttt{False} specifies to find all periodic points of period $n$
- \texttt{formal} – (default: \texttt{False}) boolean; \texttt{True} specifies to find the formal periodic points only. The formal periodic points are the points in the support of the dynatomic cycle.
- \texttt{R} – (optional) a commutative ring. Defaults to the base ring of this map.
- \texttt{algorithm} – (default: \texttt{'variety'}) must be one of the following:
  - \texttt{'variety'} - find the rational points on the appropriate variety
  - \texttt{'cyclegraph'} - find the cycles from the cycle graph
- \texttt{return\_scheme} – return a subscheme of the ambient space that defines the $n$th periodic points

OUTPUT:
A list of periodic points of this map or the subscheme defining the periodic points.

EXAMPLES:

\texttt{sage: set\_verbose(None)}
\texttt{sage: P.<x,y> = ProjectiveSpace(QQbar, 1)}
\texttt{sage: f = DynamicalSystem_projective([x^2 - x*y + y^2, x^2 - y^2 + x*y])}
\texttt{sage: f.periodic\_points(1)}
\[(-0.5000000000000000? - 0.866025403784439?*I : 1),
  (-0.5000000000000000? + 0.866025403784439?*I : 1),
  (1 : 1)]

\texttt{sage: P.<x,y,z> = ProjectiveSpace(QuadraticField(5,'t'), 2)}
\texttt{sage: f = DynamicalSystem_projective([x^2 - 21/16*z^2, y^2 - z^2, z^2])}
\texttt{sage: f.periodic\_points(2)}
(continues on next page)
\begin{verbatim}
((-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))
\end{verbatim}
sage: f = DynamicalSystem_projective([x^2 - 3/4*z^2, y^2 - 3/4*z^2, z^2])
sage: f.periodic_points(2, formal=True)
[(-1/2 : -1/2 : 1), (-1/2 : 3/2 : 1), (3/2 : -1/2 : 1)]

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)]

sage: P.<x,y> = ProjectiveSpace(GF(13^2, 't'), 1)
sage: f = DynamicalSystem_projective([x^3 + 3*y^3, x*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)]

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)]
sage: f.periodic_points(2, R=GF(7))
[]

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

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)
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)]

5.3. Dynamical systems on projective schemes
sage: f.periodic_points(1, return_scheme=True)
Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
- \(x^3 + x^2y - y^3 + x^2z^2\),
- \(-x^2y^2 + x^2z - y^2z + x^2z^2\),
- \(-y^3 + x^2z + yz^2 - z^3\)

sage: P.<x,y>=ProjectiveSpace(GF(3), 1)
sage: f = DynamicalSystem_projective([x^2 - 2*y^2, y^2])
sage: f.periodic_points(2, R=GF(3^2,'t'))
[(t + 2 : 1), (2*t : 1)]

sage: S.<c> = QQ[]
sage: R.<x,y> = PolynomialRing(S, 2)
sage: P = ProjectiveSpace(R)
sage: f = DynamicalSystem_projective([x^2 + c*y^2, y^2], return_scheme=True)
Closed subscheme of Projective Space of dimension 1 over Univariate Polynomial Ring in c over Rational Field defined by:
\(x^2 + x*y + (c + 1)*y^2\)

sage: P.<x,y,z> = ProjectiveSpace(ZZ, 2)
sage: f = DynamicalSystem([x^2 - 2*y^2, y^2, z^2])
sage: X = f.periodic_points(2, minimal=False, formal=True, return_scheme=True)

\[\text{possible_periods(**kwds)}\]
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 self.possible_periods() modulo all primes of good reduction in range prime_bound. Return the intersection of those lists.

INPUT:
kwds:

- **prime_bound** – (default: \([1, 20]\)) a list or tuple of two positive integers or an integer for the upper bound
- **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 positive integers

EXAMPLES:

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 - 29/16*y^2, y^2])

(continues on next page)
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]

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]

postcritical_set(check=True)

Return the postcritical set of this dynamical system.

Raises an error if this dynamical system is not postcritically finite.

The postcritical set is union of points which are in the forward orbits of the critical points. In other words, the set of points $Q$ such that $f^n(P) = Q$ for some positive integer $n$ and critical point $P$, where $f$ is this map.

Note that the orbit of all critical points is found, even if the critical points are defined in an extension of the base ring of this dynamical system. We extend to the field defined by $f$. field_of_definition_critical(), where $f$ is this map.

INPUT:

- check – (default: True) boolean; whether to check if this dynamical system is postcritically finite or not.

OUTPUT: The set of postcritical points.

EXAMPLES:

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem([[x^3 - 3/2* x*y^2, y^3]])
sage: f.postcritical_set(
[(1/2*a : 1), (-1/2*a : 1), (1 : 0)]

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem([[1/2*x^3 - 9/2*x*y^2, y^3]])
sage: f.postcritical_set(check=False)
[(1 : 1), (-1/2 : 1), (1 : 0)]
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem([-4*y^2, 9*x^2 - 12*x*y])
sage: f.postcritical_set()
[(1 : 1), (4/3 : 1), (1 : 0), (0 : 1)]

sage: K.<v> = QuadraticField(2)
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem([x^2 + (-2)*y^2, y^2])
sage: m = matrix(K, 2, 2, [v, 1, 0, 1])
sage: g = f.conjugate(m)
sage: g.postcritical_set()
[(-3/2*a : 1), (1/2*a : 1), (1 : 0)]

sage: F.<z> = FiniteField(9)
sage: P.<x,y> = ProjectiveSpace(F, 1)
sage: f = DynamicalSystem([x^2 + (-2)*y^2, y^2])
sage: m = matrix(F, 2, 2, [z, 1, 0, 1])
sage: g = f.conjugate(m)
sage: g.postcritical_set()
[(1 : 0), (0 : 1), (a + 2 : 1)]

preperiodic_points(m, n, **kwds)

Computes the preperiodic points of period m, n of this dynamical system defined over the ring R or the base ring of the map.

This is done by finding the rational points on the variety defining the points of period m, n.

For rational maps, where there are potentially infinitely many periodic points of a given period, you must use the return_scheme option. Note that this scheme will include the indeterminacy locus.

INPUT:

• n - a positive integer, the period
• m - a non negative integer, the preperiod

kwds:

• minimal – (default: True) boolean; True specifies to find only the preperiodic points of minimal period m,"n" and False specifies to find all preperiodic points of period m, n
• formal – (default: False) boolean; True specifies to find the formal periodic points only. The formal periodic points are the points in the support of the dynatomic cycle.
• R – (default: the base ring of the dynamical system) a commutative ring over which to find the preperiodic points
• return_scheme – (default: False) boolean; return a subscheme of the ambient space that defines the m,"n"th preperiodic points

OUTPUT:

A list of preperiodic points of this map or the subscheme defining the preperiodic points.

EXAMPLES:

sage: P.<x,y> = ProjectiveSpace(QQbar, 1)
sage: f = DynamicalSystem_projective([x^2 - y^2, y^2])

(continues on next page)
sage: f.preperiodic_points(0, 1)
[(-0.618033988749895? : 1), (1 : 0), (1.618033988749895? : 1)]

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2 - 29/16*y^2, y^2])
sage: f.preperiodic_points(1, 3)
[(-5/4 : 1), (1/4 : 1), (7/4 : 1)]

sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([x^2 - 3/4*y^2, y^2, z^2])
sage: f.preperiodic_points(0, 2, formal=True)
[(-1/2 : 1 : 0), (-1/2 : 1 : 1)]

sage: P.<x,y> = ProjectiveSpace(QQbar, 1)
sage: f = DynamicalSystem_projective([x^2 - x*y + 2*y^2, x^2 - y^2])
sage: f.preperiodic_points(1, 2, minimal=False)
[(-3.133185666641252? : 1),
 (-0.3478103847799310? - 1.028852254136693?*I : 1),
 (-0.3478103847799310? + 1.028852254136693?*I : 1),
 (0.8165928333206258? - 0.671006757431000?*I : 1),
 (0.8165928333206258? + 0.671006757431000?*I : 1),
 (1 : 0),
 (1 : 1),
 (1.695620769559862? : 1),
 (3 : 1)]

sage: R.<w> = QQ[]
sage: K.<s> = NumberField(w^6 - 3*w^5 + 5*w^4 - 5*w^3 + 5*w^2 - 3*w + 1)
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: sorted(f.preperiodic_points(0, 1), key=str)
[(-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 +
 → 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 - 3 : 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 -
 → 1 : 1),
 (1 : 1 : 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 + 6*s)^-
 → 2 : 1),
 (s^5 - 2*s^4 + 2*s^3 + s : s^5 - 3*s^4 + 4*s^3 - 3*s^2 + 2*s - 1 : 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 +
 → 2 : 1)]

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2 + 1/4*y^2, y^2])
sage: f.preperiodic_points(1, 1, formal=True)
[(-1/2 : 1), (1 : 0)]

sage: P.<x,y> = ProjectiveSpace(QQ, 1)

(continues on next page)
Discrete dynamics, Release 10.1

(continued from previous page)

```python
sage: f = DynamicalSystem_projective([x^2 - 3/4*y^2, y^2])
sage: f.preperiodic_points(0, 2, formal=True)
[(-1/2 : 1)]
```

```python
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: K.<v> = QuadraticField(5)
sage: phi = QQ.embeddings(K)[0]
sage: f = DynamicalSystem_projective([x^2 - y^2, y^2])
sage: f.preperiodic_points(1, 1, R=phi)
[(-1/2*v - 1/2 : 1), (1/2*v - 1/2 : 1)]
```

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ, 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)
sage: f.preperiodic_points(1, 1)
[(-1/4 : -1/2 : 1), (1 : 2 : 1)]
```

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([x^2, y^2, z^2])
sage: f.preperiodic_points(1, 1)
[(-3/2 : -1 : 1), (-3/2 : 1 : 1), (-1/2 : -1 : 1), (1/2 : -1 : 1),
(1/2 : 1 : 1), (3/2 : -1 : 1)]
```

When the ring over which to find the preperiodic points is a number field, the ordering of the preperiodic points might depend on the architecture (32 or 64 bits):

```python
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: R.<z> = QQ[]
sage: K.<v> = NumberField(z^4 - z^2 - 1)
sage: f = DynamicalSystem_projective([x^2 - y^2, y^2])
sage: sorted(f.preperiodic_points(2, 1, R=K), key=str)
[(-v : 1), (v : 1)]
```

Chapter 5. Arithmetic Dynamical Systems
Discrete dynamics, Release 10.1

\begin{verbatim}
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([x^2 - 3/4*y^2, y^2, z^2])
sage: f.preperiodic_points(0, 2, formal=True)
[(-1/2 : 1 : 0), (-1/2 : 1 : 1)]
sage: S.<c> = QQ[]
sage: R.<x,y> = PolynomialRing(S, 2)
sage: P = ProjectiveSpace(R)
sage: f = DynamicalSystem_projective([x^2 + c*y^2, y^2])
sage: f.preperiodic_points(1, 2, return_scheme=True)
Closed subscheme of Projective Space of dimension 1 over Univariate Polynomial Ring in c over Rational Field defined by:
x^2 - x*y + (c + 1)*y^2
\end{verbatim}

\textbf{primes_of_bad_reduction}(\texttt{check=True})

Determine the primes of bad reduction for this dynamical system.

Must be defined over a number field.

If \texttt{check} is True, each prime is verified to be of bad reduction.

\textbf{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.

\textbf{INPUT:}

\begin{itemize}
\item \texttt{check} – (default: True) boolean
\end{itemize}

\textbf{OUTPUT:} a list of primes

\textbf{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]
sage: P.<x,y> = ProjectiveSpace(QQ,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}

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}

5.3. Dynamical systems on projective schemes
This is an example where check=False returns extra primes:

```python
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)
[2, 5, 37, 2239, 304432717]
sage: f.primes_of_bad_reduction()
[5, 37, 2239, 304432717]
```

**ramification_type**\((R=\text{None}, \text{stable=True})\)

Return the ramification type of endomorphisms of \(\mathbb{P}^1\).

Only branch points defined over the ring \(R\) contribute to the ramification type if specified, otherwise \(R\) is the ring of definition for \(self\).

Note that branch points defined over \(R\) may not be geometric points if stable not set to True.

If \(R\) is specified, stable is ignored.

If stable, then this will return the ramification type over an extension which splits the Galois orbits of critical points.

**INPUT:**

- \(R\) – ring or morphism (optional)
- \(\text{split}\) – boolean (optional)

**OUTPUT:**

list of lists, each term being the list of ramification indices in the pre-images of one critical value

**EXAMPLES:**

```python
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: F = DynamicalSystem_projective([x^4, y^4])
sage: F.ramification_type()
[[4], [4]]

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: F = DynamicalSystem_projective([x^3, 4*y^3 - 3*x^2*y])
sage: F.ramification_type()
[[2], [2], [3]]

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: F = DynamicalSystem_projective([(x + y)^4, 16*x*y*(x-y)^2])
sage: F.ramification_type()
[[2], [2], [4]]

sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: F = DynamicalSystem_projective([(x + y)*(x - y)^3, y*(2*x+y)^3])
sage: F.ramification_type()
[[3], [3], [3]]

sage: F = DynamicalSystem_projective([x^3 - 2*x^2*y^2 + 2*y^3, y^3])
sage: F.ramification_type()
```

(continues on next page)
red**reduced_form**ed Form

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, first period to try to find appropriate binary form
- `emb` – (optional) embedding of base field into \( \mathbb{C} \)
- `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) # this needs 2 periodic
```

(continues on next page)
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])  # this 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
......:  + 2292*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)
(  
(continues on next page)
Dynamical System of Projective Space of dimension 1 over Rational Field
Defn: Defined on coordinates by sending (X : Y) to
\(-7X^4 - 12X^3Y - 42X^2Y^2 - 12X^2Y^3 - 7Y^4
: -X^4 - 4X^3Y - 6X^2Y^2 - 4X^2Y^3 - Y^4),
\[
\begin{bmatrix}
-1 & 2 \\
2 & -5 \\
\end{bmatrix}
\]
sage: P.<x,y> = ProjectiveSpace(RR, 1)
sage: f = DynamicalSystem_projective([x^4, RR(sqrt(2))*y^4])
sage: m = matrix(RR, 2, 2, [1,12,0,1])
sage: f = f.conjugate(m)
sage: g, m = f.reduced_form(smallest_coeffs=False); m
\[
\begin{bmatrix}
1 & -12 \\
0 & 1 \\
\end{bmatrix}
\]
sage: P.<x,y> = ProjectiveSpace(CC, 1)
sage: f = DynamicalSystem_projective([x^4, CC(sqrt(-2))*y^4])
sage: m = matrix(CC, 2, 2, [1,12,0,1])
sage: f = f.conjugate(m)
sage: g, m = f.reduced_form(smallest_coeffs=False); m
\[
\begin{bmatrix}
1 & -12 \\
0 & 1 \\
\end{bmatrix}
\]
sage: K.<w> = QuadraticField(2)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: f = DynamicalSystem_projective([x^3, w*y^3])
sage: m = matrix(K, 2, 2, [1,12,0,1])
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^2 - 2 with w = 1.414213562373095?
Defn: Defined on coordinates by sending (x : y) to (x^3 : w*y^3)
\[
\begin{bmatrix}
1 & -12 \\
0 & 1 \\
\end{bmatrix}
\]
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)
(continues on next page)
Dynamical System of Projective Space of dimension 1 over Number Field in \(w\) with defining polynomial \(x^5 + x - 3\) with \(w = 1.132997565885066?\)
Defn: Defined on coordinates by sending \((x : y)\) to
\[
\begin{pmatrix}
12^*x^3 : (2334^*w)^*y^3
\end{pmatrix},
\]
\[
\begin{bmatrix}
0 & -1 \\
1 & -12
\end{bmatrix}
\]

```
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)
,
\[
\begin{bmatrix}
2 & -2 \\
3 & 0
\end{bmatrix}
\]
)
```

```
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)
,
\[
\begin{bmatrix}
-1 & -2 \\
1 & 1
\end{bmatrix}
\]
)
```

```
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)
,
\[
\begin{bmatrix}
2 & -1 \\
0 & 2
\end{bmatrix}
\]
)
```

```
sage: P.<x,y> = QQ[]
sage: f = DynamicalSystem([4*x^2 + y^2, 4*y^2])
sage: f.reduced_form() # long time
```
(continues on next page)
Discrete dynamics, Release 10.1

Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to
  \((x^2 - x*y + y^2 : y^2)\),
  
  \[
  \begin{bmatrix}
  2 & -1 \\
  0 & 2 
  \end{bmatrix}
  \]

resultant(normalize=False)

Compute 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', return_polynomial=False, chow=False, deform=False, check=True)
Computes the values of the elementary symmetric polynomials evaluated on the \( n \) multiplier spectra of this dynamical system.

The sigma invariants are the symmetric polynomials evaluated on the characteristic polynomial of the multipliers. See [Hutz2019] for the full definition. Specifically, this function returns either the following polynomial or its coefficients (with signs appropriately adjusted):

\[
\prod_{P \text{ period } n} (w - c(P, t)),
\]

where \( c(P, t) \) is the charactereristic polynomial (variable \( t \)) of the multiplier at \( P \). Note that in dimension 1, only the coefficients of the constant term is returned.

The invariants can be computed for points of period \( n \) or points of formal period \( n \). 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). Only implemented for dimension 1. 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:**

In dimension 1, we use the Poisson product of the resultant of two polynomials:

\[
\text{res}(f, g) = \prod_{f(a)=0} g(a).
\]

In higher dimensions, we use elimination theory (Groebner bases) to compute the equivalent of the Poisson product. Letting \( f \) be the polynomial defining the periodic or formal periodic points and \( g \) the polynomial \( w - F \) for an auxiliary variable \( w \) and \( F \) the characteristic polynomial of the Jacobian matrix of \( f \). Note that if \( f \) is a rational function, we clear denominators for \( g \).

To calculate the full polynomial defining the sigma invariants, we follow the algorithm outlined in section 4 of [Hutz2019]. There are 4 cases:

- multipliers and \( n \) periodic points all distinct – in this case, we can use Proposition 4.1 of [Hutz2019] to compute the sigma invariants.
- \( n \) periodic points are all distinct, multipliers are repeated – here we can use Proposition 4.2 of [Hutz2019] to compute the sigma invariants. This corresponds to \texttt{chow=True}.
- \( n \) periodic points are repeated, multipliers are all distinct – to deal with this case, we deform the map by a formal parameter \( k \). The deformation separates the \( n \) periodic points, making them distinct, and we can recover the \( n \) periodic points of the original map by specializing \( k \) to 0. This corresponds to \texttt{deform=True}.
- \( n \) periodic points are repeated, multipliers are repeated – here we can use both cases 2 and 3 together. This corresponds to \texttt{deform=True} and \texttt{chow=True}.

As we do not want to check which case we are in beforehand, we throw a ValueError if the computed polynomial does not have the correct degree.

**INPUT:**

- \( n \) – a positive integer, the period
• **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 \( n \).

• **embedding** – (default: *None*) must be *None*, passing an embedding is no longer supported, see :trac:`32205`.

• **type** – (default: *'point'*) string; either *'point'* or *'cycle'* depending on whether you compute with one multiplier per point or one per cycle. Not implemented for dimension greater than 1.

• **return polynomial** – (default: *False*) boolean; *True* specifies returning the polynomial which generates the sigma invariants, see [Hutz2019] for the full definition. The polynomial is always a multivariate polynomial with variables \( w \) and \( t \).

• **chow** – (default: *False*) boolean; *True* specifies using the Chow algorithm from [Hutz2019] to compute the sigma invariants. While slower, the Chow algorithm does not lose information about multiplicities of the multipliers. In order to accurately compute the sigma polynomial when there is a repeated multiplier, **chow** must be *True*.

• **deform** – (default: *False*) boolean; *True* specifies first deforming the map so that all periodic points are distinct and then calculating the sigma invariants. In order to accurately calculate the sigma polynomial when there is a periodic point with multiplicity, **deform** must be *True*.

• **check** – (default: *True*) boolean; when *True* the degree of the sigma polynomial is checked against the expected degree. This is done as the sigma polynomial may drop degree if multiplicities of periodic points or multipliers are not correctly accounted for using **chow** or **deform**.

**Warning:** Setting check to *False* can lead to mathematically incorrect answers.

**OUTPUT:** a list of elements in the base ring, unless **return_polynomial**

is *True*, in which case a polynomial in \( w \) and \( t \) is returned. The variable \( t \) is the variable of the characteristic polynomials of the multipliers.

If this map is defined over \( \mathbb{P}^N \), where \( N > 1 \), then the list is the coefficients of \( w \) and \( t \), in lexicographical order with \( w > t \).

**EXAMPLES:**

```sage
P.<x,y> = ProjectiveSpace(QQ, 1)
f = DynamicalSystem_projective([x^2 + x*y + y^2, y^2 + x*y])
f.sigma_invariants(1)

[3, 3, 1]
```

If **return_polynomial** is *True*, then following [Hutz2019] we return a two variable polynomial in \( w \) and \( t \):

```sage
P.<x,y> = ProjectiveSpace(QQ, 1)
f = DynamicalSystem_projective([x^2 + 2*y^2, y^2])
sage: poly = f.sigma_invariants(1, return_polynomial=True); poly
w^3 - 3*w^2*t + 2*w^2 + 3*w*t^2 - 4*w*t + 8*w - t^3 + 2*t^2 - 8*t
```

From the full polynomial, we can easily recover the one variable polynomial whose coefficients are symmetric functions in the multipliers, up to sign:
For dynamical systems on $\mathbb{P}^N$, where $N > 1$, the full polynomial is needed to distinguish the conjugacy class. We can, however, still return a list in this case:

When calculating the sigma invariants for $\mathbb{P}^N$, with $N > 1$, the default algorithm loses information about multiplicities. Note that the following call to sigma invariants returns a degree 6 polynomial in $w$:

Setting `chow` to `True`, while much slower, accounts correctly for multiplicities. Note that the following returns a degree 7 polynomial in $w$:

check that infinity as part of a longer cycle is handled correctly:

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: 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([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)
[1957526074450617/1048576, -9078122048145044298567432325/2147483648,
 -262266111490999878224381377917540931367/1099511627776,
 -2622661107937102104196133701280271632423/549755813888,
 33852320483016116503153209450763500631714178825448006778305/72057594037927936, ...
 => 0]

sage: P.<x,y,z> = ProjectiveSpace(GF(5), 2)
sage: f = DynamicalSystem([x^2, y^2, z^2])
sage: f.sigma_invariants(1, chow=True, return_polynomial=True)
 w^7 - 2*w^6*t^2 + w^6 + w^5*t^4 + w^5*t + w^4*t^3 + 2*w^4*t^2 + w^3*t^5 - w^3*t^4 - 2*w^3*t^3 - w^2*t^10 + w^2*t^7 + w^2*t^6 + w^2*t^5 + 2*w*t^12 - w*t^10 + w*t^9 - 2*w*t^8 - w*t^7 - t^14 + 2*t^9
sage: R.<c> = QQ[]
sage: P.<x,y> = ProjectiveSpace(R, 1)
sage: f = DynamicalSystem_projective([x^2 + c*y^2, y^2])
sage: f.sigma_invariants(1, return_polynomial=True)
(continues on next page)
\[ w^3 + (-3) w^2 t + 2 w^2 + 3 w t^2 + (-4) w t + 4 c w - t^3 + 2 t^2 + (-4 c) t \]

```
sage: f.sigma_invariants(2, chow=True, formal=True, return_polynomial=True)
```

\[ w^2 + (-2) w t + (8 c + 8) w + t^2 + (-8 c - 8) t + 16 c^2 + 32 c + 16 \]

```
sage: R.<c,d> = QQ[]
sage: P.<x,y,z> = ProjectiveSpace(R, 2)
sage: f = DynamicalSystem([x^2 + c*z^2, y^2 + d*z^2, z^2])
sage: len(dict(f.sigma_invariants(1, return_polynomial=True)))
```

51

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem([x^2 + 3*y^2, x*y])
sage: f.sigma_invariants(1, deform=True, return_polynomial=True)
```

\[ w^3 - 3 w^2 t + 3 w^2 + 3 w t^2 - 6 w t + 3 w - t^3 + 3 t^2 - 3 t + 1 \]

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')
```

\[ [170, 5195, 172700, 968615, 1439066, 638125, 0] \]

class sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field(polys, do-main)

Bases: DynamicalSystem_projective, SchemeMorphism_polynomial_projective_space_field

all_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 (\( m \in \mathbb{N} \)). 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:

- **R** – (default: domain of dynamical system) the base ring over which the periodic points of the dynamical system are found
- **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
- **period_degree_bounds** – (default: [4, 4]) a pair of positive integers (max period, max degree) for which the dynatomic polynomial should be solved for
- **algorithm** – (optional) specifies which algorithm to use; current options are dynatomic and lifting; defaults to solving the dynatomic for low periods and degrees and lifts for everything else
- **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:**

```python
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 - 3/4*y^2, y^2])
sage: sorted(f.all_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.all_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 : 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)]

sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([-5*x^2 + 4*y^2, 4*x*y])
sage: sorted(f.all_periodic_points())  # long time
[(-2 : 1), (-2/3 : 1), (2/3 : 1), (1 : 0), (2 : 1)]

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: sorted(f.all_periodic_points())
[(-w + 1 : 1), (w : 1), (1 : 0)]
```

(continues on next page)

5.3. Dynamical systems on projective schemes 183
Discrete dynamics, Release 10.1

```
sage: P.<u,v> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([u^2 + v^2, u*v])
sage: f.all_periodic_points()
Traceback (most recent call last):
...
NotImplementedError: rational periodic points for number fields
only implemented for polynomials
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: K.<v> = QuadraticField(5)
sage: phi = QQ.embeddings(K)[0]
sage: f = DynamicalSystem_projective([x^2 - y^2, y^2])
sage: sorted(f.all_periodic_points(R=phi))
[(-1 : 1), (-1/2*v + 1/2 : 1), (0 : 1), (1 : 0), (1/2*v + 1/2 : 1)]
```

```
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: f = DynamicalSystem_projective([x^2 - (3/4)*w^2, y^2 - 3/4*w^2,
....:
        z^2 - 3/4*w^2, w^2])
sage: sorted(f.all_periodic_points(algorithm="dynatomic"))
[(-1/2 : -1/2 : -1/2 : 1),
 (-1/2 : -1/2 : 3/2 : 1),
 (-1/2 : 3/2 : -1/2 : 1),
 (-1/2 : 3/2 : 3/2 : 1),
 (0 : 0 : 1 : 0),
 (0 : 1 : 0 : 0),
 (0 : 1 : 1 : 0),
 (1 : 0 : 0 : 0),
 (1 : 0 : 1 : 0),
 (1 : 1 : 0 : 0),
 (1 : 1 : 1 : 0),
 (3/2 : -1/2 : -1/2 : 1),
 (3/2 : -1/2 : 3/2 : 1),
 (3/2 : 3/2 : -1/2 : 1),
 (3/2 : 3/2 : 3/2 : 1)]
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2 - 3/4*y^2, y^2])
sage: sorted(f.all_periodic_points(period_degree_bounds=[2,2]))
[(-1/2 : 1), (1 : 0), (3/2 : 1)]
```

```
all_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 \texttt{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:
• R – (default: domain of dynamical system) the base ring over which the periodic points of the dynamical system are found
• 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
• period_degree_bounds – (default: [4, 4]) a pair of positive integers (max period, max degree) for which the dynatomic polynomial should be solved for when in dimension 1
• algorithm – (optional) specifies which algorithm to use; current options are dynatomic and lifting; defaults to solving the dynatomic for low periods and degrees and lifts for everything else

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.all_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.all_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.all_preperiodic_points(prime_bound=[1,8],
                                      lifting_prime=7, periods=[2]))
 (1/4 : -2 : 1), (1/4 : -1 : 1), (1/4 : 0 : 1), (1/4 : 1 : 1), (1/4 : 2 : 1),
 (0 : -2 : 1), (0 : -1 : 1), (0 : 0 : 1), (0 : 1 : 1), (0 : 2 : 1),
 (1 : -2 : 1), (1 : -1 : 1), (1 : 0 : 1), (1 : 1 : 1), (1 : 2 : 1),
 (2 : -2 : 1), (2 : -1 : 1), (2 : 0 : 1), (2 : 1 : 1), (2 : 2 : 1)]
```

```
sage: K.<w> = QuadraticField(33)
sage: PS.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x^2 - 71/48*y^2, y^2])
```

(continues on next page)
sage: sorted(f.all_preperiodic_points())  # long time
[(-1/12*w - 1/2 : 1), (-1/6*w + 1/4 : 1), (1/12*w - 1/2 : 1), (1/12*w + 1/2 : 1), (-1/12*w + 1/2 : 1), (-1/6*w - 1/4 : 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)]

all_rational_preimages(points)

Given a set of rational points in the domain of this dynamical system, return all the rational preimages of those points.

In others 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:

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)]))

A non-periodic example

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)]
```
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: sorted(f.all_rational_preimages([P(16*w^2 - 29, 16)]), key=str)
[(-w - 1/2 : 1),
 (-w : 1),
 (-w^2 + 21/16 : 1),
 (-w^2 + 29/16 : 1),
 (-w^2 - w + 25/16 : 1),
 (-w^2 - w + 33/16 : 1),
 (w + 1/2 : 1),
 (w : 1),
 (w^2 + w - 25/16 : 1),
 (w^2 + w - 33/16 : 1),
 (w^2 - 21/16 : 1),
 (w^2 - 29/16 : 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, R=None, num_cpus=2)

Return the set of elements in PGL over the base ring that conjugates one dynamical system to the other.

Given two nonconstant rational functions of equal degree, determine if there is a rational element of PGL that conjugates one rational function to another.

The optional argument \( R \) specifies the field of definition of the PGL elements. The set is determined by taking the fixed points of one map and mapping them to permutations of the fixed points of the other map. As conjugacy preserves the multipliers as a set, fixed points are only mapped to fixed points with the same multiplier. 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.

**Warning:** For degree 1 maps that are conjugate, there is a positive dimensional set of conjugations. This function returns only one such element.

**ALGORITHM:**

Implementing invariant set algorithm from the paper [FMV2014]. Uses the set of \( n \) th preimages of fixed points, as this set is invariant under conjugation to find all elements of PGL that take one set to another. Additionally, keeps track of multiplier information to reduce the necessary combinatorics.

**INPUT:**

- **other** – a rational function of same degree as this map
- **R** – a field or embedding
- **num_cpus** – (default: 2) the number of threads to run in parallel. Increasing num_cpus can potentially greatly speed up this function.

### 5.3. Dynamical systems on projective schemes

187
OUTPUT:

Set of conjugating $n + 1$ by $n + 1$ matrices.

AUTHORS:

- Original algorithm written by Xander Faber, Michelle Manes, Bianca Viray [FMV2014].
- Implemented by Rebecca Lauren Miller as part of GSOC 2016.
- Algorithmic improvement by Alexander Galarraga as part of GSOC 2021.

EXAMPLES:

```python
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]]
```

Increasing `num_cpus` can speed up computation:

```python
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: f = DynamicalSystem_projective([x^2, y^2, z^2, w^2])
sage: len(f.conjugating_set(f, num_cpus=3))
24
```

```python
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: sorted(f.conjugating_set(g))
[[[-1 -1]
  [ 2 1]],
 [[ 1 1]],
 [[ 2 1], [2 1]]]
```

```python
sage: K.<i> = QuadraticField(-1)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: D8 = DynamicalSystem_projective([y^3, x^3])
sage: sorted(D8.conjugating_set(D8))
[[[-1 0] [-i 0] [ 0 -1] [ 0 -i] [0 i] [0 1] [i 0] [1 0]]
 [[ 0 1], [ 0 i], [1 0], [1 i], [1 0], [0 1], [0 i], [0 1]]]
```

```python
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):
...
```

(continues on next page)
ValueError: no more rational preimages; try extending the base field and trying again

```
sage: P.<x,y> = ProjectiveSpace(GF(7), 1)
sage: D6 = DynamicalSystem_projective([y^2, x^2])
sage: sorted(D6.conjugating_set(D6))
```
```
[ [0 1] [0 2] [0 4] [1 0] [2 0] [4 0]
[1 0], [1 0], [1 0], [0 1], [0 1], [0 1]
```

```
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)
```
```
[ [1 0 0]
[0 1 0]
[0 0 1]
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: R = P.coordinate_ring()
sage: f = DynamicalSystem_projective([R(3), R(4)])
sage: g = DynamicalSystem_projective([R(5), R(2)])
sage: m = f.conjugating_set(g)[0]
sage: f.conjugate(m) == g
```
```
True
```

```
sage: P.<x,y,z> = ProjectiveSpace(GF(11), 2)
sage: m1 = matrix(GF(11), 3, 3, [1,4,1,0,2,1,1,1,1])
sage: g = f.conjugate(m1)
sage: f.conjugating_set(g)
```
```
[ [ 1 0 0]
[ 9 1 4]
[ 4 10 8]
```

```
sage: L.<v> = CyclotomicField(8)
sage: P.<x,y,z> = ProjectiveSpace(L, 2)
sage: f = DynamicalSystem_projective([2*v^x + 12*v^y, 11*v^y + 2*v*z, x + z])
```

5.3. Dynamical systems on projective schemes 189
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:**

```python
sage: R.<x> = PolynomialRing(QQ)
sage: K.<w> = NumberField(x^3 + 1/4*x^2 - 41/16*x + 23/64)
sage: PS.<x,y> = ProjectiveSpace(1,K)
sage: f = DynamicalSystem_projective([x^2 - 29/16*y^2, y^2])
sage: P = PS([w,1])
sage: sorted(f.connected_rational_component(P), key=str)
[(-w - 1/2 : 1),
 (-w : 1),
 (-w^2 + 21/16 : 1),
 (-w^2 + 29/16 : 1),
 (-w^2 - w + 25/16 : 1),
 (-w^2 - w + 33/16 : 1),
 (w + 1/2 : 1),
 (w : 1),
 (w^2 + w - 25/16 : 1),
 (w^2 + w - 33/16 : 1),
 (w^2 - 21/16 : 1),
 (w^2 - 29/16 : 1)]
```

```python
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),
 (17/16 : -7/4 : 1),
 (1386468673/4294967296 : -81343/65536 : 1),
 (-47/256 : -17/16 : 1),
 (47/256 : -17/16 : 1)],
```
Discrete dynamics, Release 10.1

(continued from previous page)

(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, R=None, num_cpus=2)`

Return whether two dynamical systems are conjugate over their base ring (by default) or over the ring \( R \) entered as an optional parameter.

**ALGORITHM:**

Implementing invariant set algorithm from the paper [FMV2014]. Uses the set of \( n \) th preimages of fixed points, as this set is invariant under conjugation to find all elements of PGL that take one set to another. Additionally, keeps track of multiplier information to reduce the necessary combinatorics.

**INPUT:**

- `other` – a nonconstant rational function of the same degree as this map
- `num_cpus` – (default: 2) the number of threads to run in parallel. Increasing `num_cpus` can potentially greatly speed up this function.

**OUTPUT:** boolean

**AUTHORS:**

- Original algorithm written by Xander Faber, Michelle Manes, Bianca Viray [FMV2014].
- Implemented by Rebecca Lauren Miller as part of GSOC 2016.
- Algorithmic improvement by Alexander Galarraga as part of GSOC 2021.

**EXAMPLES:**

```python
sage: K.<w> = CyclotomicField(3)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: D8 = DynamicalSystem_projective([y^2, x^2])
sage: D8.is_conjugate(D8)
True
```

We can speed up computation by increasing `num_cpus`:

```python
sage: P.<x,y,z,w> = ProjectiveSpace(QQ,3)
sage: f = DynamicalSystem_projective([x^2, y^2, z^2, w^2])
sage: f.is_conjugate(f, num_cpus=2)
True
```

```python
sage: set_verbose(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)
True
```

5.3. Dynamical systems on projective schemes
Discrete dynamics, Release 10.1

```
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(QQbar, 1)
sage: f = DynamicalSystem_projective([7*x + 12*y, 8*x])
sage: g = DynamicalSystem_projective([1645*x - 318*y, 8473*x - 1638*y])
sage: f.is_conjugate(g)
True

Conjugation is only checked over the base field by default:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([-3*y^2, 3*x^2])
sage: g = DynamicalSystem_projective([-x^2 - 2*x*y, 2*x*y + y^2])
sage: f.is_conjugate(g), f.is_conjugate(g, R=QQbar)
(False, True)

sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([7*x + 12*y, 8*y + 2*z, x + z])
sage: m1 = matrix(QQ, 3, 3, [1,4,1,0,2,1,1,1,1])
sage: g = f.conjugate(m1)
sage: f.is_conjugate(g)
True

sage: P.<x,y,z> = ProjectiveSpace(GF(7), 2)
sage: f = DynamicalSystem_projective([2*x + 12*y, 11*y + 2*z, x + z])
sage: m1 = matrix(GF(7), 3, 3, [1,4,1,0,2,1,1,1,1])
sage: g = f.conjugate(m1)
sage: f.is_conjugate(g)
True

sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([2*x^2 + 12*y*x, 11*y*x + 2*y^2, x^2 + z^\rightarrow 2])
sage: m1 = matrix(QQ, 3, 3, [1,4,1,0,2,1,1,1,1])
```

(continues on next page)
is_newton(return_conjugation=False)

Return whether self is a Newton map.

A map $g$ is Newton if it is conjugate to a map of the form $f(z) = z - \frac{p(z)}{p'(z)}$ after dehomogenization, where $p(z)$ is a squarefree polynomial.

INPUT:

* return_conjugation – (default: False) if the map is Newton and True, then return the conjugation that moves this map to the above form

OUTPUT:

A Boolean. If return_conjugation is True, then this also returns the conjugation as a matrix if self is Newton or None otherwise.

The conjugation may be defined over an extension if the map has fixed points not defined over the base field.

EXAMPLES:

```python
sage: A.<z> = AffineSpace(QQ, 1)
sage: f = DynamicalSystem_affine([z - (z^2 + 1)/(2*z)])
sage: F = f.homogenize(1)
sage: F.is_newton(return_conjugation=True)
(True, [1 0]

sage: A.<z> = AffineSpace(QQ, 1)
sage: f = DynamicalSystem_affine([z^2 + 1])
sage: F = f.homogenize(1)
sage: F.is_newton(return_conjugation=True)
(False, None)

sage: PP.<x,y> = ProjectiveSpace(QQ, 1)
sage: F = DynamicalSystem_projective([-4*x^3 - 3*x*y^2, -2*y^3])
sage: F.is_newton(return_conjugation=True)[1]
[-4*a 2*a]
```

sage: K.<zeta> = CyclotomicField(2*4)
sage: A.<z> = AffineSpace(K, 1)
sage: f = DynamicalSystem_affine(z-(z^3+zeta*z)/(3*z^2+zeta))
sage: F = f.homogenize(1)
sage: F.is_newton()  # True
"
**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
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
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
```

```sage
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
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
sage: PS.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([6*x**2 + 12*x*y + 7*y**2, 12*x**2*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:

```python
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.

```python
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), 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^2 + 45*y^2 - 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:

```python
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^2 + 2*x*y - 5*x^2, 2*x*y])
```

(continues on next page)
sage: f.normal_form()
Traceback (most recent call last):
...
NotImplementedError: 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 w3 of size 3^3
  Defn: Defined on coordinates by sending (x : y) to
        (x^3 + x^2*y + x*y^2 + (-w3)*y^3 : y^3)

sage: P.<x,y> = ProjectiveSpace(GF(3),1)
sage: f = DynamicalSystem_projective([2*x^3 + x^2*y, y^3])
sage: g,m,psi = f.normal_form(return_conjugation= True); psi
Ring morphism:
  From: Finite Field of size 3
  To:   Finite Field in z2 of size 3^2
  Defn: 1 |---> 1

potential_good_reduction(prime, return_conjugation=False)

Return True if this dynamical system has potential good reduction at prime.

A dynamical system has good reduction at prime if after the coefficients are reduced modulo prime the degree remains the same. A dynamical system $f$ has potential good reduction if there exists $\phi \in PGL(n, \overline{K})$ such that $\phi^{-1} \circ f \circ \phi$ has good reduction.

If this dynamical system $f$ has potential good reduction at prime, a dynamical system $g = \phi^{-1} \circ f \circ \phi$ which has good reduction at prime is returned.

This dynamical system must have as its domain $\mathbb{P}^1(\mathbb{K})$, where $\mathbb{K}$ is a number field.

INPUT:
Discrete dynamics, Release 10.1

• prime – a prime ideal of the field of definition of the fixed points of the map, or a prime number in \( \mathbb{Q} \)
  if the field of definition of the fixed points is \( \mathbb{Q} \).

• return_conjugation – (default: False) if set to True, the \( PGL_2 \) map used to achieve good reduction will be returned

OUTPUT:

A tuple:

• The first element is: - False if this dynamical system does not have potential good reduction. - True if this dynamical system does have potential good reduction.

• The second element is: - None if this dynamical system does not have potential good reduction. - A dynamical system with good reduction at prime otherwise.

• If return_conjugation is True, then the tuple will have a third element, which is: - None if this dynamical system does not have potential good reduction. - The \( PGL_2 \) map used to achieve good reduction otherwise.

EXAMPLES:

```sage
P.<x,y> = ProjectiveSpace(QQ, 1)
sage: system = DynamicalSystem_projective([x^2 - y^2, 2*x*y])
sage: prime = system.field_of_definition_periodic(1).prime_above(2)
sage: new_system = system.potential_good_reduction(prime)
```

```
Dynamical System of Projective Space of dimension 1 over Number Field in a with defining polynomial x^2 + 1
  Defn: Defined on coordinates by sending (x : y) to
        ((-1/2*a)*x^2 + (-5/2*a)*y^2 : (-a)*x*y + y^2)
```

Note that this map has good reduction at 2:

```sage
dSystem(resultant())
1
```

Using return_conjugation, we can get the conjugation that achieves good reduction:

```sage
cconj = system.potential_good_reduction(prime, True)
```

```
[-1/2*a 1/2]
[ 0 1]
```

We can check that this conjugation achieves good reduction:

```sage
dSystem.conjugate(conj).resultant()
1
```

```sage:
P.<x,y> = ProjectiveSpace(QQ, 1)
sage: system = DynamicalSystem_projective([x^5 - x*y^4, 5*y^5])
sage: prime = system.field_of_definition_periodic(1).prime_above(5)
```

(continues on next page)
rational_preperiodic_graph(**kwds)

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 \( P^1 \), 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:**

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 digraph representing the orbits of the rational preperiodic points in projective space.

**EXAMPLES:**

```python
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
```

```python
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
```

```python
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], # long time 
.....: lifting_prime=13)
Looped digraph on 30 vertices
```

(continued from previous page)
sage: K.<w> = QuadraticField(-3)
sage: P.<x,y> = ProjectiveSpace(K,1)
sage: f = DynamicalSystem_projective([x^2 + y^2, y^2])
sage: f.rational_preperiodic_graph()  # long time
Looped digraph on 5 vertices

reduce_base_field()

Return this map defined over the field of definition of the coefficients.

The base field of the map could be strictly larger than the field where all of the coefficients are defined. This function reduces the base field to the minimal possible. This can be done when the base ring is a number field, $\mathbb{Q}$, a finite field, or algebraic closure of a finite field.

OUTPUT: A dynamical system

EXAMPLES:

sage: K.<t> = GF(2^3)
sage: P.<x,y,z> = ProjectiveSpace(K, 2)
sage: f = DynamicalSystem_projective([x^2 + y^2, y^2, z^2+z*y])
sage: f.reduce_base_field()
Dynamical System of Projective Space of dimension 2 over Finite Field of size 2
  Defn: Defined on coordinates by sending $(x : y : z)$ to $(x^2 + y^2 : y^2 : y*z + z^2)$

sage: P.<x,y,z> = ProjectiveSpace(QQbar, 2)
sage: f = DynamicalSystem_projective([x^2 + QQbar(sqrt(3))*y^2, y^2, QQbar(sqrt(2))*z^2])
sage: f.reduce_base_field()
Dynamical System of Projective Space of dimension 2 over Number Field in a with defining polynomial $y^4 - 4*y^2 + 1$ with a = -0.5176380902050415?
  Defn: Defined on coordinates by sending $(x : y : z)$ to $(x^2 + (-a^2 + 2)*y^2 : y^2 : (a^3 - 3*a)*z^2)$

sage: R.<x> = QQ[]
sage: K.<v> = NumberField(x^3 - 2, embedding=(x^3 - 2).roots(ring=CC)[0][0])
sage: L.<w> = NumberField(x^6 + 9*x^4 - 4*x^3 + 27*x^2 + 36*x + 31, embedding=(x^6 + 9*x^4 - 4*x^3 + 27*x^2 + 36*x + 31).roots(ring=CC)[0][0])
sage: P.<x,y> = ProjectiveSpace(L,1)
sage: f = DynamicalSystem([L(v)*x^2 + y^2, x*y])
sage: f.reduce_base_field().base_ring().is_isomorphic(K)
True

sage: K.<v> = CyclotomicField(5)
sage: A.<x,y> = ProjectiveSpace(K, 1)
sage: f = DynamicalSystem_projective([3*x^2 + y^2, x*y])
sage: f.reduce_base_field()
Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending $(x : y)$ to $(3*x^2 + y^2 : x*y)$
class sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_finite_field(polys, domain)

Bases: DynamicalSystem_projective_field, SchemeMorphism_polynomial_projective_space_finite_field

all_periodic_points(**kwds)

Return a list of all periodic points over a finite field.

INPUT:

keywords:

• R – (default: base ring of dynamical system) the base ring over which the periodic points of the dynamical system are found

OUTPUT: a list of elements which are periodic

EXAMPLES:

```python
sage: P.<x,y> = ProjectiveSpace(GF(5^2),1)
sage: f = DynamicalSystem_projective([x^2 + y^2, x*y])
sage: f.all_periodic_points()
[(1 : 0), (z2 + 2 : 1), (4*z2 + 3 : 1)]
```

```python
sage: P.<x,y,z> = ProjectiveSpace(GF(5),2)
sage: f = DynamicalSystem_projective([x^2 + y^2 + z^2, x*y + x*z, z^2])
sage: f.all_periodic_points()
[(1 : 0 : 0),
 (0 : 0 : 1),
 (1 : 0 : 1),
 (2 : 1 : 1),
 (1 : 4 : 1),
 (3 : 0 : 1),
 (0 : 3 : 1)]
```

```python
sage: P.<x,y> = ProjectiveSpace(GF(3^2, 't'))
sage: f = DynamicalSystem_projective([x^2 - y^2, y^2])
sage: f.all_periodic_points(R=GF(3^2, 't'))
[(1 : 0), (0 : 1), (2 : 1), (t : 1), (2*t + 1 : 1)]
```

automorphism_group(**kwds)

Return the subgroup of $PGL_2$ that is the automorphism group of this dynamical system.

The automorphism group is the set of $PGL_2$ elements that fixed the map under conjugation.

For dimension 1, see [FMV2014] for the algorithm.

For dimension greater than 1, we compute the conjugating set of this dynamical system with itself.

INPUT:

The following keywords are used when the dimension of the domain is greater than 1:

• num_cpus – (default: 2) the number of threads to use. Setting to a larger number can greatly speed up this function.

The following keywords are used when the dimension of the domain is 1:
• 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 \( PGL_2 \) 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: lst, label = f.automorphism_group(return_functions=True, # long time
... iso_type=True)
```

```
sage: sorted(lst, key=str), label # long time
([(2*x + 1)/(x + 1),
  (2*x + 1)/x,
  (2*x + 2)/(x + 2),
  (2*x + 2)/x,
  (x + 1)/(x + 2),
  (x + 1)/x,
  (x + 2)/(x + 1),
  (x + 2)/x,
  1/(x + 1),
  1/(x + 2),
  1/x,
  2*x,
  2*x + 1,
  2*x + 2,
  2*x/(x + 1),
  2*x/(x + 2),
  2/(x + 1),
  2/(x + 2),
  2/x,
  x,
  x + 1,
  x + 2,
  x/(x + 1),
  x/(x + 2)],
'PGL(2,3)')
```
Discrete dynamics, Release 10.1

```python
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')
```

```python
sage: R.<x,y> = ProjectiveSpace(GF(3^4,'t'),1)
sage: f = DynamicalSystem_projective([x^2 + 25*x*y + y^2, x*y + 3*y^2])
sage: f.automorphism_group(absolute=True)
[Univariate Polynomial Ring in w over Finite Field in b of size 3^4, ]
[ [1 0]
  [0 1]
]
```

```python
sage: R.<x,y,z> = ProjectiveSpace(GF(5), 2)
sage: f = DynamicalSystem_projective([x^3 + x*z^2, y^3 + y*z^2, z^3])
sage: all([f.conjugate(m) == f for m in f.automorphism_group()])
True
```

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:

```python
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
```

```python
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
```

```python
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
```

```python
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
```

```python
sage: P.<x,y,z> = ProjectiveSpace(GF(3),2)
sage: X = P.subscheme([x - y])
```

(continues on next page)
Discrete dynamics, Release 10.1

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

is_postcritically_finite(**kwds)
Every point is postcritically finite in a finite field.

INPUT: None. kwds is to parallel the overridden function

OUTPUT: the boolean True

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.is_postcritically_finite()
True

sage: P.<x,y> = ProjectiveSpace(GF(13),1)
sage: f = DynamicalSystem_projective([x^4 - x^2*y^2 + y^4, y^4])
sage: f.is_postcritically_finite(use_algebraic_closure=False)
True

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 tuple (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^4 - x^2*y^2 + y^4, y^4])
(2, 3)

sage: R.<t> = GF(13^3)
sage: P.<x,y> = ProjectiveSpace(R,1)
possible_periods\( (\text{return\_points}=\text{False}) \)

Return the list of possible minimal periods of a periodic point over \( \mathbb{Q} \) and optionally a point in each cycle.

ALGORITHM:
See [Hutz2009].

INPUT:

- \( \text{return\_points} \) – (default: \text{False}) boolean; if \text{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 \text{return\_points} is \text{True}.

EXAMPLES:

```sage
sage: P.<x,y> = ProjectiveSpace(GF(23),1)
sage: f = DynamicalSystem_projective([x^2 - 2*y^2, y^2])
sage: f.possible_periods()
[1, 5, 11, 22, 110]
```

```sage
sage: P.<x,y> = ProjectiveSpace(GF(13),1)
sage: f = DynamicalSystem_projective([x^2 - y^2, y^2])
sage: sorted(f.possible_periods(\text{True}))
[[[0 : 1], 2], [[1 : 0], 1], [[3 : 1], 3], [[3 : 1], 36]]
```

```sage
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

5.4 Dynamical systems for products of projective spaces

This class builds on the product projective space class. The main constructor functions are given by \text{DynamicalSystem} and \text{DynamicalSystem\_projective}. The constructors function can take either polynomials or a morphism from which to construct a dynamical system.

The must be specified.

EXAMPLES:
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).

class sage.dynamics.arithmetic_dynamics.product_projective_ds.DynamicalSystem_product_projective(polys,
domain)

Bases: DynamicalSystem, ProductProjectiveSpaces_morphism_ring

The class of dynamical systems on products of projective spaces.

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:

• polys – a list of n_1 + \cdots + n_r multi-homogeneous polynomials, all of which should have the same parent

• domain – a projective scheme embedded in P^{n_1-1} \times \cdots \times P^{n_r-1}

EXAMPLES:

sage: T.<x,y,z,w,u> = ProductProjectiveSpaces([2, 1], QQ)
sage: f = 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).

nth_iterate(P, n, normalize=False)
Return the n-th iterate of 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?

INPUT:

• P – a point in self.domain()

• n – a positive integer

• normalize – (default: False) boolean

OUTPUT: A point in self.codomain()

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]) (continues on next page)
Discrete dynamics, Release 10.1

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 nth 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, x^2, y^2, z^2], 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),..., F^N(P)].

If N is a list or tuple N = [m,k] return [F^m(P),...,F^k(P)]. Automatically normalize the points if normalize is True. Perform the checks on point initialize if check is True.

INPUT:

• P – a point in self.domain()

• N – a non-negative integer or list or tuple of two non-negative integers

kwds:

• check – (default: True) boolean

• normalize – (default: False) boolean

OUTPUT: a list of points in 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), (1/2 : 1 , 1 : 0 : 1), (1/12 : 1 , -1 : 1 : 0), (1/1872 : 1 , 1 : 1 : 0)]
```

```python
sage: Z.<a,b,x> = 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)
[(1 : 3 , 1 : 2), (1 : 3 , -7 : 4), (1 : 3 , 407 : 112), (1 : 3 , 66014215 : 5105408)]
```

class sage.dynamics.arithmetic_dynamics.product_projective_ds.DynamicalSystem_product_projective_field(polys, domain)

Bases: DynamicalSystem_product_projective

class sage.dynamics.arithmetic_dynamics.product_projective_ds.DynamicalSystem_product_projective_finite_field(polys, domain)

Bases: DynamicalSystem_product_projective_field

cyclegraph()

Return the digraph of all orbits of this morphism mod $p$.

OUTPUT: a digraph

EXAMPLES:

```python
sage: P.<a,b,c,d> = ProductProjectiveSpaces(GF(3), [1,1])
sage: f = DynamicalSystem_projective([a^2,b^2,c^2,d^2], domain=P)
sage: f.cyclegraph()
Looped digraph on 16 vertices
```

```python
sage: P.<a,b,c,d> = ProductProjectiveSpaces(GF(5), [1,1])
sage: f = DynamicalSystem_projective([a^2,b^2,c,d], domain=P)
sage: f.cyclegraph()
Looped digraph on 36 vertices
```

```python
sage: P.<a,b,c,d,e> = ProductProjectiveSpaces(GF(2), [1,2])
sage: f = DynamicalSystem_projective([a^2,b^2,c,d,e], domain=P)
sage: f.cyclegraph()
Looped digraph on 21 vertices
```

Todo: Dynamical systems for subschemes of product projective spaces needs work. Thus this is not implemented for subschemes.

5.4. Dynamical systems for products of projective spaces 207
5.5 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(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: WehlerK3Surface_ring

EXAMPLES:

```
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
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: WehlerK3Surface_ring

class sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_finite_field(polys)
Bases: WehlerK3Surface_field

cardinality()
Counts the total number of points on the K3 surface.

ALGORITHM:
Enumerate points over $\mathbb{P}^2$, 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*x2*y0*y2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
```

(continues on next page)
sage: X.cardinality()
55

class sage.dynamics.arithmetic_dynamics.wehlerK3.WeheK3Surface_ring(polys)
Bases: AlgebraicScheme_subscheme_product_projective

A K3 surface in P^2 × 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)

Return the G polynomials G^*_k.

They are defined as: G^*_k = (L^*_i) Q^*_i - L^*_i L^*_j Q^*_ij + (L^*_i)^2 Q^*_ij, 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: 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*y2 + x1^2*y2^2
sage: X = WehlerK3Surface([Z, Y])
sage: X.Gpoly(1, 0)
x0^2*y1^2 + x1^4 - x0*x1^2*y2 + x1^3*y2 + x1^2*y2^2 + x2^4

Hpoly(component, i, j)

Return the H polynomials defined as H^*_ij.

This polynomial is defined by:

H^*_ij = 2L^*_i L^*_j Q^*_ik - L^*_i L^*_j Q^*_jk - L^*_i L^*_k Q^*_ij + (L^*_i)^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:

\[
\begin{align*}
\text{sage: } & R.<x0,x1,x2,y0,y1,y2> = PolynomialRing(ZZ, 6) \\
\text{sage: } & Y = x0^2*y0 + x1*y1 - x2*y2 \\
\text{sage: } & Z = x0^2*y0*y1 + x0^2*y2^2 - x0*x1*y1*y2 + x1^2*y2*y1 \\
& \quad + x2^2*y2*y2 + x2^2*y1*y2 + x1^2*y2*y2 \\
\text{sage: } & X = WehlerK3Surface([Z, Y]) \\
\text{sage: } & X.Hpoly(0, 1, 0) \\
& 2*y0*y1^3 + 2*y0*y1*y2^2 - y1*y2^3 \\
\end{align*}
\]

\(L_{xa}(a)\)

Function will return the \(L\) polynomial defining the fiber, given by \(L_{xa}\).

This polynomial is defined as:

\[L_{xa} = \{(a, y) \in \mathbb{P}^2 \times \mathbb{P}^2 : L(a, y) = 0\}.\]

Notation and definition from: [CS1996]

INPUT: \(a\) - Point in \(\mathbb{P}^2\)

OUTPUT: A polynomial representing the fiber

EXAMPLES:

\[
\begin{align*}
\text{sage: } & PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ) \\
\text{sage: } & Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 \\
& \quad + 3*x0*x1*y0*y1 - 2*x2^2*y0*y1 - \ \\
& x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 - 4*x1*x2*y1^2 \\
& \quad + 5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 \\
& \quad + 4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2 \\
\text{sage: } & Y = x0*y0 + x1*y1 + x2*y2 \\
\text{sage: } & X = WehlerK3Surface([Z, Y]) \\
\text{sage: } & T = PP(1, 1, 0, 1, 0, 0) \\
\text{sage: } & X.Lxa(T[0]) \\
& y0 + y1
\end{align*}
\]

\(L_{yb}(b)\)

Function will return a fiber by \(L_{yb}\).

This polynomial is defined as:

\[L_{yb} = \{(x, b) \in \mathbb{P}^2 \times \mathbb{P}^2 : L(x, b) = 0\}.\]

Notation and definition from: [CS1996]

INPUT: \(b\) - Point in projective space

OUTPUT: A polynomial representing the fiber

EXAMPLES:

\[
\begin{align*}
\text{sage: } & PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ) \\
\text{sage: } & Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0*y1 \\
& \quad + 3*x0*x1*y0*y1 - 2*x2^2*y0*y1 - \ \\
& - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 - 4*x1*x2*y1^2 \\
& \quad + 5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 \\
& \quad + 4*x1^2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2 \\
\text{sage: } & Y = x0*y0 + x1*y1 + x2*y2 \\
\text{sage: } & X = WehlerK3Surface([Z, Y]) \\
\text{sage: } & T = PP(1, 1, 0, 1, 0, 0) \\
\text{sage: } & X.Lxa(T[0]) \\
y0 + y1
\end{align*}
\]
sage: Y = x0*y0 + x1*y1 + x2*y2
sage: X = WehlerK3Surface([Z, Y])
sage: T = PP(1, 1, 0, 1, 0, 0)
sage: X.Lyb(T[1])
\[x0\]

\textbf{Qxa}(a)

Function will return the \( Q \) polynomial defining a fiber given by \( Q_a \).

This polynomial is defined as:
\[ Q_a = \{(a, y) \in \mathbb{P}^2 \times \mathbb{P}^2 : Q(a, y) = 0\}. \]

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 - 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.Qxa(T[0])
\[5*y0^2 + 7*y0*y1 + y1^2 + 11*y1*y2 + y2^2\]

\textbf{Qyb}(b)

Function will return a fiber by \( Q_b \).

This polynomial is defined as:
\[ Q_b = \{(x, b) \in \mathbb{P}^2 \times \mathbb{P}^2 : Q(x, b) = 0\}. \]

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 + 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:

$$g^* = \frac{(H_i)^2 - 4G_iG_j}{(L_i)}.$$  

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
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*y0^2*y1 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0^2*x2*y1^2 - 4*x1^2*y0*y2 + 5*x0*x2*y0^2*y2
\rightarrow 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.Ramification_poly(0)
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*y1*y2^2 - 80*y0^3*y1*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*y2^3 + 8*y0*y1*y2^3 - 112*y1^2*y2^3 + y2^6
```

**Sxa(a)**

Function will return fiber by $S_a^x$.

This function is defined as:

$$S_a^x = L_a^x \cap Q_a^x.$$  

Notation and definition from: [CS1996]

**INPUT:** a - Point in $\mathbb{P}^2$

**OUTPUT:** A subscheme 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 + 3*x0*x1*y0^2*y1 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0^2*x2*y1^2 - 4*x1^2*y0*y2 + 5*x0*x2*y0^2*y2
\rightarrow 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
\rightarrow defined by:
```

(continues on next page)
y0 + y1,  
5*y0^2 + 7*y0*y1 + y1^2 + 11*y1*y2 + y2^2

\textbf{Syb}(b)

Function will return fiber by \( S^b \).

This function is defined by:
\[
S^b = L^b \cap Q^b.
\]

Notation and definition from: [CS1996]

INPUT: \( b \) - Point in \( \mathbb{P}^2 \)

OUTPUT: A subscheme representing the fiber

\textbf{EXAMPLES}:

\begin{verbatim}
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*x1^2*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
\end{verbatim}

\textbf{canonical_height} \((P, N, badprimes=None, prec=100)\)

Evaluates the canonical height for \( P \) with \( N \) terms of the series of the local heights.

\textbf{ALGORITHM}:

The sum of the canonical height minus and canonical height plus, for more info see section 4 of [CS1996].

INPUT:

\begin{itemize}
  \item \( P \) – a surface point
  \item \( N \) – positive integer (number of terms of the series to use)
  \item \( \text{badprimes} \) – (optional) list of integer primes (where the surface is degenerate)
  \item \( \text{prec} \) – (default: 100) float point or p-adic precision
\end{itemize}

OUTPUT: A real number

\textbf{EXAMPLES}:

\begin{verbatim}
sage: set_verbose(None)
sage: R.<x0,x1,x2,y0,y1,y2> = PolynomialRing(QQ, 6)
sage: L = (-y0 - y1)*x0 + (-y0^2*x1 + y2^2*x2)
sage: Q = (-y2^2*y0^2 - y1^2)*x0^2 + ((-y0^2 - y2^2*y0 + (-y2^2*y1 - y2^2))*x1 + \
    (-y0^2 - y2^2*y1)*x2)*x0 + ((-y0^2 - y2^2*y0 - y2^2)*x1^2 + (-y2^2*y0 - y1^2)*x2^2*x1 + \
    + (-y0^2 + (-y1 - y2)*y0)*x2^2)
\end{verbatim}
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:

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(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)
- \( \text{badprimes} \) – (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:

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\n    + (-y0^2 - y2*y1)*x2)*x0 + ((-y0^2 - y2*y0 - y2^2)*x1^2 + (-y2*y0 - y1^...
    - 2)*x2*x1\n    + (-y0^2 + (-y1 - y2)*y0)*x2^2)
sage: X = WehlerK3Surface([L, Q])
sage: P = X([1, 0, -1, 1, -1, 0])  # order 16
sage: X.canonical_height_minus(P, 5)  # long time
0.00000000000000000000000000000

Call-Silverman example:
Discrete dynamics, Release 10.1

```
sage: setVerbose(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 +\n    3*x0*x1*y0*y1 - 2*x2^2*y0*y1 + x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2^2*y1^2 -\n    4*x1^2*y1^2 + 5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 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: P = X([0, 1, 0, 0, 0, 1])
sage: X.canonical_height_minus(P, 4) # long time
0.55073705369676788175590206734
```

canonical_height_plus(P, N, badprimes=None, 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
- 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:**

```
sage: setVerbose(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 +\n    (-y0^2 - y2*y0 - y2^2)*x2)*x0 + ((-y0^2 + (-y1 - y2)*y0)*x2^2 +\n    (-y0^2 + (-y1 - y2)*y0)*x2^2)

sage: X = WehlerK3Surface([L, Q])
sage: P = X([1, 0, -1, 1, -1, 0]) #order 16
sage: X.canonical_height_plus(P, 5) # long time
0.00000000000000000000000000000
```

Call-Silverman Example:

```
sage: setVerbose(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 +\n    3*x0*x1*y0*y1 - 2*x2^2*y0*y1 + x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2^2*y1^2 -\n    4*x1^2*y1^2 + 5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 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])
```

(continues on next page)
sage: P = X([0, 1, 0, 0, 0, 1])
sage: X.canonical_height_plus(P, 4) # long time
0.14752753298983071394400412161

\textbf{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:**

\begin{verbatim}
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
\end{verbatim}

\textbf{degenerate\_fibers()}

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 $\text{self}.Gpoly(1, 0)$, $\text{self}.Gpoly(1, 1)$, $\text{self}.Gpoly(1, 2)$, $\text{self}.Hpoly(1, 0, 1)$, $\text{self}.Hpoly(1, 0, 2)$, $\text{self}.Hpoly(1, 1, 2)$ (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:**

\begin{verbatim}
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])
sage: X.degenerate_fibers()
[[]], [(1 : 0 : 0)]
\end{verbatim}

\begin{verbatim}
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*x0*x2*y0*y1 - x0*x2*y1^2 + 2*x1*x2*y1^2 - x0*x2*y1^2 - 4*x1*x2*y1^2 + 5*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0*x1*y2^2 + 4*x1*x2*y1*y2 + x0*x1*y2^2 + 3*x2^2*y2^2
sage: Y = x0*y0 + x1*y1 + x2*y2
\end{verbatim}
Discrete dynamics, Release 10.1

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 + (y0^2 + (-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 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)*y0 + (2*y0^2 + y2*y0 + (2*y1^2 + y2^2))*x1 + (2*y1^2 + y2*y0 + (2*y1^2 + y2^2))*x2)*x0 + ((2*y0^2 + \\
+ (y1^2 + 2*y2)*y0 + (2*y0^2 + y2*y0 + (2*y1^2 + y2^2))*x1^2 + (2*y0^2 + y2*y0 + (2*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)

Return the fibers $[y \text{ (component = 1)} \text{ or } x \text{ (Component = 0)}]$ of a point on a K3 Surface.

This 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*x0^2*x1 + y0^2*x2^2 - y0*y1*x1*y2 + y1^2*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*y2 - 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*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 - 1*x2*y2
sage: Q=(2*x0^2 + x2*x0 + (2*x1^2 + x2^2))*y0^2 + ((x0^2 + x1*x0 +(x1^2 +
         2*x2*x1 + x2^2))*y1 +
         (2*x1^2 + x2*x1 + x2^2)*y2)*y0 + ((2*x0^2+ (x1 + 2*x2)*x0 + (2*x1^2 +
         x2*x1 + 2*x2^2))*y1^2 +
         (2*x1 + 2*x2)*x0 + 
         (x1^2 +x2*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*y0^2 + x0^2*x1*y0^2 + x1^2*y0^2 + x0^2*y0*y1 + x1^2*y0*y1 -
         x0*x1*y1*y2 + y1^2*x2^2 + y2^2*x1^2 + y1^2*x2^2
sage: X = WehlerK3Surface([Z, Y])
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 -
  \rightarrow \ \ 
2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 -4*x1*x2*y1^2 +
\rightarrow \ \ 
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 -
  \rightarrow \ \ 
2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 -4*x1*x2*y1^2 +
\rightarrow \ \ 
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*x0*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 -4*x1*x2*y1^2 \ 
-4*x0*x2^2*y0*y2 + 5*x0*x2^2*y0*y2 - x0^2*y2^2 + 4*x1^2*y2^2 + 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^2*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 = x0^2*y0 + x1^2*y1 + x2^2*y2
sage: Q = x0^2*y0^2 + 2*x2^2*y0*y1 + x0^2*y1^2 - x0^2*y1^2
sage: W2 = WehlerK3Surface([L, Q])
sage: W1.is_isomorphic(W2)
False

5.5. Wehler K3 Surfaces 219
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*y1 + x0^2*y2^2 - x0*x1*y1*y2 + x1^2*y2^2 + 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: Z = x0^2*y0*y2 + 3*x0*x1*y0^2 + x1^2*y0*y2 + 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_smooth()  
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*y2 + 3*x0*x1*y0^2 + x1^2*y0*y2 + 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: orbit = X.orbit_psi(T, 4)  
sage: X.is_symmetric_orbit(orbit)  
True  
```

```
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)  
sage: L = x0*y0 + x1*y1 + x2*y2  
(continues on next page)"
\[
\text{sage: } Q = x_1^2 y_0^2 + 2 x_2^2 y_0^2 y_1 + x_0^2 y_1^2 - x_0 x_1 y_2^2
\]
\[
\text{sage: } W = \text{WehlerK3Surface([}L, Q\text{])}
\]
\[
\text{sage: } T = W([-1, -1, 1, 1, 0, 1])
\]
\[
\text{sage: } \text{Orb = } W.\text{orbit}_\phi(T, 7)
\]
\[
\text{sage: } W.\text{is_symmetric_orbit(Orb)}
\]
False

\textbf{lambda\_minus}(P, v, N, m, n, 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:

\[
\text{sage: } PP.<x_0,x_1,x_2,y_0,y_1,y_2> = \text{ProductProjectiveSpaces([}2, 2\text{], QQ)}
\]
\[
\text{sage: } Z = x_0^2 y_0^2 + 3 x_0 x_1 y_0^2 + x_1^2 y_0^2 + 4 x_0 y_0 y_1 + 3 x_0 x_1 y_1 + 1 \\ 
- 2 x_2^2 y_0 y_1 - x_0^2 y_1^2 + 2 x_1^2 y_1^2 - x_0 x_2 y_1^2 - 4 x_1 x_2 y_1^2 + \ldots
\]
\[
\text{sage: } Y = x_0 y_0 + x_1 y_1 + x_2 y_2
\]
\[
\text{sage: } X = \text{WehlerK3Surface([}Z, Y\text{])}
\]
\[
\text{sage: } P = X([0, 0, 1, 1, 0, 0])
\]
\[
\text{sage: } X.\text{lambda\_minus}(P, 2, 20, 2, 0, 200)
\]
\[
-0.1857351672047135037172805779671791488351056677474271893705
\]

\textbf{lambda\_plus}(P, v, N, m, n, 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

5.5. Wehler K3 Surfaces 221
• $v$ – 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:

```
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 + 2*x0*x2*y0*y2 - 4*x1*x2*y0*y2 + 7*x0^2*y1^2 + 4*x1^2*y1^2 + x0*y1^2 + x2*y2 + 4*x1^2*y2^2 + 3*x1^2*y2^2

sage: X = WehlerK3Surface([Z, Y])
sage: P = X([0, 0, 1, 1, 0, 0])
sage: X.lambda_plus(P, 0, 10, 2, 0)
0.89230705169161608922595928129
```

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:

• check - (default: True) boolean checks to see if point is on the surface

• 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)
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: W.nth_iterate_phi(T, 7)
(-1 : 0 : 1 , 1 : -2 : 1)
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: W.nth_iterate_phi(T, -7)
(1 : 0 : 1 , -1 : 2 : 1)
```
Discrete dynamics, Release 10.1

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 - y1 - y2*y0)*x2)*x0 \
    + ((-y0*y2 - 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])
sage: X.nth_iterate_phi(P, 8) == X.nth_iterate_psi(P, 8)
True

nth_iterate_psi$(P, n, **kwds)$

Computes the nth iterate for the psi function.

**INPUT:**
- P – a point in $\mathbb{P}^2 \times \mathbb{P}^2$
- n – an integer

**kwds:**
- check – (default: True) boolean, checks to see if point is on the surface
- normalize – (default: False) boolean, normalizes the point

**OUTPUT:** The nth iterate of the point given the psi function (if $n$ is positive), or the phi function (if $n$ is negative)

**EXAMPLES:**

```
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: W.nth_iterate_psi(T, -7)
(-1 : 0 : 1 , 1 : -2 : 1)
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: W.nth_iterate_psi(T, 7)
(1 : 0 : 1 , -1 : 2 : 1)
```

orbit_phi$(P, N, **kwds)$

Return the orbit of the $\phi$ function defined by $\phi = \sigma_y \circ \sigma_x$.

This function is defined in [CS1996].

**INPUT:**
- P - Point on the K3 surface
- N - a non-negative integer or list or tuple of two non-negative integers

**kwds:**
• check – (default: True) boolean, checks to see if point is on the surface
• normalize – (default: False) boolean, normalizes the point

OUTPUT: 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*y2 + 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.orbit_psi(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_psi(T,[2,3], normalize = True)
```

orbit_psi(P, N, **kwds)

Return the orbit of the $\psi$ function defined by $\psi = \sigma_x \circ \sigma_y$.

This 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:

• check - (default: True) boolean, checks to see if point is on the surface
• 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*y2 + 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)
```

(continues on next page)
phi(a,**kwds)
Evaluates the function φ = σ_y ◦ σ_x.

ALGORITHM:
Refer to Section 6: “An algorithm to compute σ_x, σ_y, φ, and ψ” 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:

```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 - 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.phi(T)
(-1 : 0 : 1 , 0 : 1 : 0)
```

psi(a,**kwds)
Evaluates the function ψ = σ_x ◦ σ_y.

ALGORITHM:
Refer to Section 6: “An algorithm to compute σ_x, σ_y, φ, and ψ” 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:
Discrete dynamics, Release 10.1

```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*x0^2*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.psi(T)
(0 : 0 : 1 , 0 : 1 : 0)
```

sigmaX(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:**

- 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:**

```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*x0^2*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:

```python
sage: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: l = y0*x0 + y1*x1 + (y0 - y1)^2*x2
sage: q = (y1*y0)*x0^2 + ((y0^2*x1 + (y0^2 + (y1^2 - y2^2))*x2)^2*x0^2 +
    (y2*y0 + y1^2)*x1*x2 + (y0^2 + (-y1^2 + y2^2))*x2*x1
sage: X = WehlerK3Surface([l, q])
sage: X.sigmaX(X([1, 0, 0, 0, 1, -2]))
(1 : 0 : 0 , 0 : 1/2 : 1)
sage: X.sigmaX(X([1, 0, 0, 0, 0, 1]))
```

(continues on next page)
Case where we cannot distinguish the two points:

\[
\begin{align*}
\text{sage: } & \quad \text{PP.<y0,y1,y2,x0,x1,x2>=ProductProjectiveSpaces([2, 2], GF(3))} \\
& \quad \text{sage: } \quad l = x0*y0 + x1*y1 + x2*y2 \\
& \quad \text{sage: } \quad q=-3*x0^2*y0^2 + 4*x0*x1*y0^2 - 5*x0*x2*y0^2 - 5*x0*y0*y1 + \\
& \quad \quad 14*x1*x2*y0*y1 + 5*x2^2*y0*y1 - x0^2*y1^2 - 6*x0*x1*y1^2 - 6*x0*x2*y1^2 + 2*x0*x2*y1^2 + 4*x0*x2*y1^2 - \\
& \quad \quad 4*x2^2*y1^2 + 4*x0^2*y0*y2 - x1^2*y0*y2 + 3*x0*x2*y0*y2 + 6*x1*x2*y0*y2 - \\
& \quad \quad 6*x0*x1*y1*y2 - x0*y1*y1*y2 - x1^2*y1*y2 + 51*x0*x2*y1*y2 - 7*x1*x2*y1*y2 - \\
& \quad \quad 9*x2^2*y1*y2 - x0^2*y2*y2 - 4*x0*x1*y2^2 + 4*x1^2*y2^2 - x0*x2*y2^2 + \\
& \quad \quad 13*x1*x2*y2^2 + x2^2*y2^2
\end{align*}
\]

\[
\begin{align*}
\text{sage: } & \quad \text{X = WehlerK3Surface([l, q])} \\
\text{sage: } & \quad \text{P = X([1, 0, 0, 0, 1, 1])} \\
\text{sage: } & \quad \text{X.sigmaX(X.sigmaX(P))}
\end{align*}
\]

Traceback (most recent call last):
...
ValueError: cannot distinguish points in the degenerate fiber

\text{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 \(\sigma_x, \sigma_y, \phi, \text{and } \psi\)” 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:

\[
\begin{align*}
\text{sage: } & \quad \text{PP.<x0,x1,x2,y0,y1,y2>=ProductProjectiveSpaces([2, 2], QQ)} \\
\text{sage: } & \quad \text{Z = x0^2*y0^2 + 3*x0*x1*y0^2 + x1^2*y0^2 + 4*x0^2*y0^2 + y1^2 + \\
& \quad \quad 3*x0*x1*y0*y1 - 2*x2^2*y0*y1 - x0^2*y1^2 + 2*x1^2*y1^2 - x0*x2*y1^2 - \\
& \quad \quad 4*x1*x2*y1^2 + 5*x0*x2*y0*y2 + 4*x1*x2*y0*y2 + 7*x0^2*y1*y2 + 4*x1^2*y1*y2 + \\
& \quad \quad + x0*x1*y2^2 + 3*x2^2*y2^2
\end{align*}
\]
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: PP.<x0,x1,x2,y0,y1,y2> = ProductProjectiveSpaces([2, 2], QQ)
sage: l = y0*x0 + x1*y1 + (y0 - y1)*x2
sage: q = (y1*y0)*x0^2 + ((y0^2)*x1 + (y0^2 - y2^2))*x2)*x0 +\n  (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*x1*y0*y1 -\n  190*x0*x1*y0*y1 \
- 5*x1*x2*y0*y1 + 5*x0*x2*y0*y1 + 14*x1*x2*y0*y1 + 5*x2^2*y0*y1 - x0^2*y1^2 -\n  6*x0*x1*y1^2 \
- 2*x1^2*y1^2 + 2*x0*x2*y1^2 + 4*x0^2*y0*y2 - x1^2*y0*y2 +\n  3*x0*x2*y0*y2 \n+ 6*x1^2*y0*y2 - 6*x0^2*y1*y2 - 4*x0*x1*y1*y2 - x1^2*y1*y2 + 51*x0*x2*y1*y2 -\n  7*x1*x2*y1*y2 \n- 9*x0*x2*y1*y2 - x0^2*y0*y2 - 4*x0*x1*y2*y2 + 4*x1^2*y2*y2 - x0^2*y2*y2 +\n  13*x1*x2*y2*y2 - x2^2*y2*y2
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: \texttt{WehlerK3Surface\_ring}

EXAMPLES:
5.6 Dynamical systems on Berkovich space over $\mathbb{C}_p$.

A dynamical system on Berkovich space over $\mathbb{C}_p$ is determined by a dynamical system on $\mathbb{A}^1(\mathbb{C}_p)$ or $\mathbb{P}^1(\mathbb{C}_p)$, which naturally induces a dynamical system on affine or projective Berkovich space.

For an exposition of dynamical systems on Berkovich space, see chapter 7 of [Ben2019], or for a more involved exposition, chapter 2 of [BR2010].

AUTHORS:

- Alexander Galarraga (August 14th, 2020): initial implementation

```python
sage: PP.<x0, x1, x2, y0, y1, y2> = ProductProjectiveSpaces([2, 2], GF(3))
sage: w = random_WehlerK3Surface(PP)
sage: type(w)
<class 'sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_finite_field_' with category>
```

```python
5.6. Dynamical systems on Berkovich space over $\mathbb{C}_p$.  229
```

```python
sage: P.<x,y> = ProjectiveSpace(Qp(3), 1)
sage: f = DynamicalSystem_projective([2*x^2 + 4*y^2, 3*x^2 + 9*y^2])
sage: DynamicalSystem_Berkovich(f)
Dynamical system of Projective Berkovich line over $\mathbb{Q}_p$ of precision 20 induced by
\rightarrow the map
Defn: Defined on coordinates by sending (x : y) to
((2 + O(3^20))*x^2 + (1 + 3 + O(3^20))*y^2 : (3 + O(3^21))*x^2 + (3^2 + O(3^22))*y^2)
```

Or directly from polynomials:

```python
sage: P.<x,y> = ProjectiveSpace(Qp(3), 1)
sage: DynamicalSystem_Berkovich([x^2 + y^2, y^2])
(continues on next page)
```
Dynamical system of Projective Berkovich line over Cp(3) of precision 20 induced by the map
Defn: Defined on coordinates by sending \((x : y)\) to 
\((x^2 + y^2 : y^2)\)

\textit{DynamicalSystem\_Berkovich} defaults to projective:

\begin{verbatim}
sage: R.<x,y> = Qp(3)[]
sage: DynamicalSystem\_Berkovich([x^2, y^2])
\end{verbatim}

To create an affine dynamical system on Berkovich space, pass an affine dynamical system to \textit{DynamicalSystem\_Berkovich}:

\begin{verbatim}
sage: A.<z> = AffineSpace(Qp(3), 1)
sage: f = DynamicalSystem\_affine(z^2 + 1)
sage: DynamicalSystem\_Berkovich(f)
\end{verbatim}

\textit{domain} can be used to specify the type of dynamical system:

\begin{verbatim}
sage: A.<z> = AffineSpace(Qp(3), 1)
sage: C = Berkovich\_Cp\_Affine(3)
sage: DynamicalSystem\_Berkovich([z^2 + 1], C)
\end{verbatim}

We can create dynamical systems which act on Berkovich spaces backed by number fields:

\begin{verbatim}
sage: R.<z> = QQ[]
sage: A.<a> = NumberField(z^2 + 1)
sage: ideal = A.prime\_above(2)
sage: P.<x,y> = ProjectiveSpace(A, 1)
sage: B = Berkovich\_Cp\_Projective(P, ideal)
sage: DynamicalSystem\_Berkovich([x^2 + y^2, 2*a*x*y], B)
\end{verbatim}

We can use the optional parameter \texttt{ideal} to create the same dynamical system more efficiently:

\begin{verbatim}
sage: R.<z> = QQ[]
sage: A.<a> = NumberField(z^2 + 1)
sage: prime\_ideal = A.prime\_above(2)
\end{verbatim}
Discrete dynamics, Release 10.1

(continued from previous page)

\begin{verbatim}
sage: P.<x,y> = ProjectiveSpace(A, 1)
sage: DynamicalSystem_Berkovich([x^2 + y^2, 2*a*x*y], ideal=prime_ideal)
Dynamical system of Projective Berkovich line over Cp(2), with base Number Field
in a with defining polynomial z^2 + 1 induced by the map
  Defn: Defined on coordinates by sending (x : y) to
  (x^2 + y^2 : (2*a)*x*y)

Creating a map on Berkovich space creates the Berkovich space it acts on:

\begin{verbatim}
sage: P.<x,y> = ProjectiveSpace(Qp(3), 1)
sage: f = DynamicalSystem_projective([x^2, y^2])
sage: g = DynamicalSystem_Berkovich(f)
sage: B = g.domain(); B
Projective Berkovich line over Cp(3) of precision 20
\end{verbatim}

The image of type I point is the image of the center:

\begin{verbatim}
sage: P.<x,y> = ProjectiveSpace(Qp(3), 1)
sage: F = DynamicalSystem_Berkovich([x^2, y^2])
sage: B = F.domain()
sage: Q1 = B(2)
sage: F(Q1)
Type I point centered at (1 + 3 + O(3^20) : 1 + O(3^20))
\end{verbatim}

For type II/III points with no poles in the corresponding disk, the image is the type II/III point corresponding to the image of the disk:

\begin{verbatim}
sage: Q2 = B(0, 3)
sage: F(Q2)
Type II point centered at (0 : 1 + O(3^20)) of radius 3^2
\end{verbatim}

The image of any type II point can be computed:

\begin{verbatim}
sage: g = DynamicalSystem_projective([x^2 + y^2, x*y])
sage: G = DynamicalSystem_Berkovich(g)
sage: Q3 = B(0, 1)
sage: G(Q3)
Type II point centered at (0 : 1 + O(3^20)) of radius 3^0
\end{verbatim}

The image of type III points can be computed has long as the corresponding disk contains no poles of the dynamical system:

\begin{verbatim}
sage: Q4 = B(1/9, 1.5)
sage: G(Q4)
Type III point centered at (3^-2 + 3^2 + O(3^18) : 1 + O(3^20))
of radius 1.50000000000000
\end{verbatim}

Sometimes, however, the poles are contained in an extension of Q_p that Sage does not support:

\begin{verbatim}
sage: H = DynamicalSystem_Berkovich([x*y^2, x^3 + 20*y^3])
sage: H(Q4) # not tested
Traceback (most recent call last):
...
NotImplementedError: cannot check if poles lie in type III disk
\end{verbatim}

5.6. Dynamical systems on Berkovich space over \( \mathbb{C}_p \).
Q4, however, does not contain any poles of \( H \) (this can be checked using pencil and paper or the number field functionality in Sage). There are two ways around this error: the first and simplest is to have \( H \) act on a Berkovich space backed by a number field:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: B = Berkovich_Cp_Projective(P, 3)
sage: H = DynamicalSystem_Berkovich([x*y^2, x^3 + 20*y^3], B)
sage: Q4 = B(1/9, 1.5)
sage: H(Q4)
Type III point centered at (81/14581 : 1) of radius 0.00205761316872428
```

Alternatively, if checking for poles in the disk has been done already, `type_3_pole_check` can be set to `False`:

```
sage: P.<x,y> = ProjectiveSpace(Qp(3), 1)
sage: H = DynamicalSystem_Berkovich([x*y^2, x^3 + 20*y^3])
sage: B = H.domain()
sage: Q4 = B(1/9, 1.5)
sage: H(Q4, False)
Type III point centered at (3^4 + 3^{10} + 2*3^{11} + 2*3^{13} + 2*3^{14} + 2*3^{15} + 3^{17} + 2*3^{18} + 2*3^{19} + 3*20 + 3*21 + 3*22 + O(3^24) : 1 + O(3^20)) of radius 0.00205761316872428
```

**WARNING:** Setting `type_3_pole_check` to `False` can lead to mathematically incorrect answers.

**as_scheme_dynamical_system()**

Return this dynamical system as `DynamicalSystem`.

**OUTPUT:** An affine or projective `DynamicalSystem`.

**EXAMPLES:**

```
sage: P.<x,y> = ProjectiveSpace(Qp(3), 1)
sage: f = DynamicalSystem_Berkovich([x^2 + y^2, x*y])
sage: f.as_scheme_dynamical_system()
Dynamical System of Projective Space of dimension 1 over 3-adic Field with capped relative precision 20
  Defn: Defined on coordinates by sending (x : y) to (x^2 + y^2 : x*y)
```

**base_ring()**

The base ring of this dynamical system, that is, the field of definition of the coefficients.

**OUTPUT:** A field.

**EXAMPLES:**

```
sage: P.<x,y> = ProjectiveSpace(Qp(3), 1)
sage: f = DynamicalSystem_Berkovich([x^2 + y^2, y^2])
sage: f.base_ring()
3-adic Field with capped relative precision 20
```

```
sage: R.<z> = QQ[]
sage: A.<a> = NumberField(z^3 + 20)
sage: P.<x,y> = ProjectiveSpace(A, 1)
sage: f = DynamicalSystem_Berkovich([x^2, x^2 + y^2], ideal=A.prime_above(2))
```
Discrete dynamics, Release 10.1

(continued from previous page)

```python
defining_polynomials()
Return the defining polynomials.
OUTPUT: A tuple of polynomials that defines the dynamical system.
EXAMPLES:
```
```python
sage: P.<x,y> = ProjectiveSpace(Qp(3), 1)
sage: f = DynamicalSystem_projective([[2*3^2 + 4*y^2, 3*x^2 + 9*y^2]])
sage: g = DynamicalSystem_Berkovich(f)
sage: g.defining_polynomials()
((2 + O(3^20))*x^2 + (1 + 3 + O(3^20))*y^2,
(3 + O(3^21))*x^2 + (3^2 + O(3^22))*y^2)
```

domain()
Return the domain of this dynamical system.
OUTPUT: A Berkovich space over \( \mathbb{C}_p \).
EXAMPLES:
```
```
```
```
```
```
```
```
class sage.dynamics.arithmetic_dynamics.berkovich_ds.DynamicalSystem_Berkovich_affine(dynamical_system, domain)
Bases: DynamicalSystem_Berkovich
A dynamical system of the affine Berkovich line over \( \mathbb{C}_p \).
INPUT:
- dynamical_system – A DynamicalSystem_affine of relative dimension 1.
- domain – (optional) affine or projective Berkovich space over \( \mathbb{C}_p \). If the input to dynamical_system is not defined over \( \mathbb{Q}_p \) or a finite extension, domain must be specified.
EXAMPLES:
A dynamical system of the affine Berkovich line is induced by a dynamical system on \( \mathbb{Q}_p \) or an extension of \( \mathbb{Q}_p \):
```
```
```
```
```
```
```
Dynamical system can be created from a morphism:
Discrete dynamics, Release 10.1

```python
sage: H = End(A)
sage: phi = H([x + 3])
sage: DynamicalSystem_Berkovich(phi)
Dynamical system of Affine Berkovich line over Cp(5) of precision 20 induced by the → map
  Defn: Defined on coordinates by sending (x) to
           (x + 3 + O(5^20))
```

**homogenize**(n)

Return the homogenization of this dynamical system.

For dynamical systems of Berkovich space, this is the dynamical system of projective Berkovich space induced by the homogenization of the dynamical system.

**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:** a dynamical system on projective Berkovich space

**EXAMPLES:**

```python
sage: A.<x> = AffineSpace(Qp(3), 1)
sage: f = DynamicalSystem_affine(1/x)
sage: f = DynamicalSystem_Berkovich(f)
sage: f.homogenize(1)
Dynamical system of Projective Berkovich line over Cp(3) of precision 20 → induced by the map
  Defn: Defined on coordinates by sending (x0 : x1) to
           (x1 : x0)
```

```python
class sage.dynamics.arithmetic_dynamics.berkovich_ds.DynamicalSystem_Berkovich_projective(dynamical_system=None, domain=None)

Bases: DynamicalSystem_Berkovich

A dynamical system on projective Berkovich space over \( \mathbb{C}_p \).

A dynamical system on projective Berkovich space over \( \mathbb{C}_p \) is determined by a dynamical system on \( A^1(\mathbb{C}_p) \) or \( P^1(\mathbb{C}_p) \), which naturally induces a dynamical system on affine or projective Berkovich space.

**INPUT:**

- dynamical_system – a DynamicalSystem_Projective of relative dimension 1. If this input is not defined over a p-adic field, then domain MUST be specified.

- domain – (optional) projective Berkovich space over \( \mathbb{C}_p \). If the input to dynamical_system is not defined over a p-adic field, domain must be specified.

**EXAMPLES:**

We can easily create a dynamical system on Berkovich space using a dynamical system on projective space over \( \mathbb{Q}_p \):

```python
sage: P.<x,y> = ProjectiveSpace(Qp(3), 1)
sage: f = DynamicalSystem_projective([1/2*x^2 + x*y + 3*y^2, 3*x^2 + 9*y^2])
sage: DynamicalSystem_Berkovich(f)
```
```
Dynamical system of Projective Berkovich line over $\mathbb{C}_p(3)$ of precision 20 induced by the map

Defn: Defined on coordinates by sending $(x : y)$ to
\[
(2 + 3 + 3^2 + 3^3 + 3^4 + 3^5 + 3^6 + 3^7 + 3^8 + 3^9 + 3^{10} + 3^{11} + 3^{12} + 3^{13} + 3^{14} + 3^{15} + 3^{16} + 3^{17} + 3^{18} + 3^{19} + O(3^{20}))^x^2 + x^2 \cdot y + (3 + O(3^{21}))^2 + (3 + O(3^{21}))^y^2 : (3 + O(3^{21}))^x^2 + (3^2 + O(3^{22}))^y^2)
\]

Or from a morphism:

```python
sage: P1.<x,y> = ProjectiveSpace(Qp(3), 1)
sage: H = End(P1)
sage: DynamicalSystem_Berkovich(H([y, x]))
```

Dynamical system of Projective Berkovich line over $\mathbb{C}_p(3)$ of precision 20 induced by the map

Defn: Defined on coordinates by sending $(x : y)$ to
$(y : x)$

Or from polynomials:

```python
sage: P.<x,y> = ProjectiveSpace(Qp(3), 1)
sage: DynamicalSystem_Berkovich([x^2+y^2, y^2])
```

Dynamical system of Projective Berkovich line over $\mathbb{C}_p(3)$ of precision 20 induced by the map

Defn: Defined on coordinates by sending $(x : y)$ to
$(x^2 + y^2 : y^2)$

**conjugate**($M$, adjugate=False, new_ideal=None)

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.
- adjugate – (default: False) boolean, also classically called adjoint, takes a square matrix $M$ and finds the transpose of its cofactor matrix. Used for conjugation in place of inverse when specified 'True'. Functionality is the same in projective space.
- new_ideal – (optional) an ideal of the base ring of $M$. Used to specify an extension in the case where $M$ is not defined over the same number field as this dynamical system.

**OUTPUT:** a dynamical system.

**EXAMPLES:**

```python
sage: P.<x,y> = ProjectiveSpace(Qp(3), 1)
sage: f = DynamicalSystem_projective([x^2 + y^2, 2*y^2])
sage: g = DynamicalSystem_Berkovich(f)
sage: g.conjugate(Matrix([[1, 1], [0, 1]]))
```

Dynamical system of Projective Berkovich line over $\mathbb{C}_p(3)$ of precision 20 induced by the map

Defn: Defined on coordinates by sending $(x : y)$ to
$(x^2 + (2 + O(3^{20}))^x \cdot y + (3 + O(3^{21}))^2 : (3 + 0(3^{21}))^x^2 + (3^2 + 0(3^{22}))^y^2)$
Discrete dynamics, Release 10.1

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_Berkovich([x^2 + y^2, y^2], ideal=5)
sage: R.<z> = QQ[]
sage: A.<a> = NumberField(z^2 + 1)
sage: conj = Matrix([[1, a], [0, 1]])
sage: f.conjugate(conj)
Dynamical system of Projective Berkovich line over Cp(5), with base Number Field in a with defining polynomial z^2 + 1 induced by the map
  Defn: Defined on coordinates by sending (x : y) to
        (x^2 + (2*a)*x*y + (-a)*y^2 : y^2)

We can use new_ideal to specify a new domain when the base ring of M and of this dynamical system are not the same:

```
sage: ideal = A.ideal(5).factor()[1][0]; ideal
Fractional ideal (2*a + 1)
sage: g = f.conjugate(conj, new_ideal=ideal)
sage: g.domain().ideal()
Fractional ideal (2*a + 1)
```

dehomogenize(n)

Return the map induced by the standard dehomogenization.

The dehomogenization is done at the n[0] coordinate of the domain and the n[1] coordinate of 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: A dynamical system on affine Berkovich space.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(Qp(3), 1)
sage: f = DynamicalSystem_projective([x^2 + y^2, x*y + y^2])
sage: g = DynamicalSystem_Berkovich(f)
sage: g.dehomogenize(1)
Dynamical system of Affine Berkovich line over Cp(3) of precision 20 induced by
  \( \frac{x^2 + 1 + O(3^{20})}{x + 1 + O(3^{20})} \)
```

normalize_coordinates()

Normalizes the coordinates of the inducing map.

OUTPUT: None.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(Qp(3), 1)
sage: f = DynamicalSystem_Berkovich([2*x^2, 2*y^2])
sage: f.normalize_coordinates(); f
Dynamical system of Projective Berkovich line over Cp(3) of precision 20 induced by the map
```

(continues on next page)
Defn: Defined on coordinates by sending \((x : y)\) to 
\((x^2 : y^2)\)

Normalize_coordinates may sometimes fail over p-adic fields:

```sage
sage: g = DynamicalSystem_Berkovich([2*x^2, x*y])
sage: g.normalize_coordinates()  #not tested
```
Traceback (most recent call last):
...
TypeError: unable to coerce since the denominator is not 1

To fix this issue, create a system on Berkovich space backed by a number field:

```sage
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: B = Berkovich_Cp_Projective(P, 3)
sage: g = DynamicalSystem_Berkovich([2*x^2, x*y], B)
sage: g.normalize_coordinates(); g
Dynamical system of Projective Berkovich line over Cp(3), with base Rational Field induced by the map
Defn: Defined on coordinates by sending \((x : y)\) to 
\((2*x : y)\)
```
Discrete dynamics, Release 10.1

```sage
sage: P.<x,y> = ProjectiveSpace(Qp(3), 1)
sage: f = DynamicalSystem_Berkovich([x^2, y^2])
sage: f.scale_by(x); f
Dynamical system of Projective Berkovich line over Cp(3) of precision 20 →
induced by the map
Defn: Defined on coordinates by sending (x : y) to
(x^3 : x*y^2)

sage: Q.<z> = QQ[]
sage: A.<a> = NumberField(z^3 + 20)
sage: ideal = A.prime_above(3)
sage: P.<x,y> = ProjectiveSpace(A, 1)
sage: B = Berkovich_Cp_Projective(P, ideal)
sage: f = DynamicalSystem_Berkovich([x^2 + y^2, 2*x*y], B)
sage: f.scale_by(2); f
Dynamical system of Projective Berkovich line over Cp(3), with base Number
Field in a with defining polynomial z^3 + 20 induced by the map
Defn: Defined on coordinates by sending (x : y) to
(2*x^2 + 2*y^2 : 4*x*y)
```

See also:

- `sage.schemes.affine.affine_morphism`
- `sage.schemes.projective.projective_morphism`
- `sage.schemes.product_projective.morphism`
INDICES AND TABLES

- Index
- Module Index
- Search Page
sage.dynamics.arithmetic_dynamics.affine_ds, 116
sage.dynamics.arithmetic_dynamics.berkovich_ds, 229
sage.dynamics.arithmetic_dynamics.generic_ds, 109
sage.dynamics.arithmetic_dynamics.product_projective_ds, 204
sage.dynamics.arithmetic_dynamics.projective_ds, 127
sage.dynamics.arithmetic_dynamics.wehlerK3, 208
sage.dynamics.cellular_automata.catalog, 1
sage.dynamics.cellular_automata.elementary, 1
sage.dynamics.cellular_automata.glca, 9
sage.dynamics.cellular_automata.solitons, 12
sage.dynamics.complex_dynamics.mandel_julia, 31
sage.dynamics.finite_dynamical_system, 39

sage.sandpiles.sandpile, 53
add_random() (sage.sandpiles.sandpile.SandpileConfig method), 81
add_random() (sage.sandpiles.sandpile.SandpileDivisor method), 89
admissible_partitions() (in module sage.sandpiles.sandpile), 102
affine_preperiodic_model() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 130
all_k_config() (sage.sandpiles.sandpile.Sandpile method), 58
all_k_div() (sage.sandpiles.sandpile.Sandpile method), 59
all_minimal_models() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 132
all_periodic_points() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 184
all_preperiodic_points() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 184
all_rational_preimages() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 186
arakelov_zhang_pairing() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_WehlerK3 method), 232
as_scheme_dynamical_system() (sage.dynamics.arithmetic_dynamics.berkovich_ds.DynamicalSystem_Berkovich method), 111
as_scheme_morphism() (sage.dynamics.arithmetic_dynamics.generic_ds.DynamicalSystem_Generic method), 215
automorphism_group() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 135
automorphism_group() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_finite_field method), 208
ideal() (sage.sandpiles.sandpile.Sandpile method), 66
identity() (sage.sandpiles.sandpile.Sandpile method), 66
in_degree() (sage.sandpiles.sandpile.Sandpile method), 67
invariant_factors() (sage.sandpiles.sandpile.Sandpile method), 67
inverse_evolution() (sage.dynamics.finite_dynamical_system.InvertibleDiscreteDynamicalSystem method), 49
inverse_evolution_default() (sage.dynamics.finite_dynamical_system.InvertibleDiscreteDynamicalSystem method), 49
InvertibleDiscreteDynamicalSystem (class in sage.dynamics.finite_dynamical_system), 47
InvertibleFiniteDynamicalSystem (class in sage.dynamics.finite_dynamical_system), 50
is_alive() (sage.sandpiles.sandpile.SandpileDivisor method), 93
is_conjugate() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 191
is_degenerate() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 218
is_dynamical_belyi_map() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 150
is_homomesic() (sage.dynamics.finite_dynamical_system.DiscreteDynamicalSystem_complex_dynamics.mandel_julia), 34
is_invariant() (sage.dynamics.finite_dynamical_system.FiniteDynamicalSystem method), 47
is_isomorphic() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 219
is_linearly_equivalent() (sage.sandpiles.sandpile.SandpileDivisor method), 94
is_newton() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 193
is_PGL_minimal() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 149
is_polynomial() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 193
is_postcritically_finite() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 151
is_postcritically_finite() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 203
is_q_reduced() (sage.sandpiles.sandpile.SandpileDivisor method), 94
is_recurrent() (sage.sandpiles.sandpile.SandpileConfig method), 86
is_smooth() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 203
is_symmetric() (sage.sandpiles.sandpile.Sandpile method), 219
is_stable() (sage.sandpiles.sandpile.SandpileConfig method), 86
is_superstable() (sage.sandpiles.sandpile.SandpileConfig method), 86
is_symmetric() (sage.sandpiles.sandpile.SandpileDivisor method), 95
is_symmetric_orbit() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 220
is_undirected() (sage.sandpiles.sandpile.Sandpile method), 67
is_weierstrass_pt() (sage.sandpiles.sandpile.SandpileDivisor method), 95
jacobian_representatives() (sage.sandpiles.sandpile.Sandpile method), 67
julia_plot() (in module sage.dynamics.complex_dynamics.mandel_julia), 32
kneading_sequence() (in module sage.dynamics.complex_dynamics.mandel_julia), 34
lambda_minus() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 221
lambda_plus() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 221
laplacian() (sage.sandpiles.sandpile.Sandpile method), 68
latex_state_evolution() (sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata method), 22
latex_states() (sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata method), 23
lift_to_rational_periodic() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_field method), 194
Lxa() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 203
Lyb() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 210
mandelbrot_plot() (in module sage.dynamics.complex_dynamics.mandel_julia),
markov_chain() (sage.sandpiles.sandpile.Sandpile method), 69
max_stable() (sage.sandpiles.sandpile.Sandpile method), 70
max_stable_div() (sage.sandpiles.sandpile.Sandpile method), 70
max_superstables() (sage.sandpiles.sandpile.Sandpile method), 70
min_cycles() (in module sage.sandpiles.sandpile), 104
min_recurrents() (sage.sandpiles.sandpile.Sandpile method), 71
minimal_model() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 152
module
sage.dynamics.arithmetic_dynamics.affine_ds
sage.dynamics.arithmetic_dynamics.berkovich_ds
sage.dynamics.arithmetic_dynamics.cellular_automata
sage.dynamics.arithmetic_dynamics.complex_dynamics
sage.dynamics.arithmetic_dynamics.generic_ds
sage.dynamics.arithmetic_dynamics.projective_ds
sage.dynamics.arithmetic_dynamics.product_projective_ds
sage.dynamics.arithmetic_dynamics.affine_ds
sage.dynamics.finite_dynamical_system
sage.dynamics.cellular_automata
sage.dynamics.complex_dynamics
sage.dynamics.arithmetic_dynamics
sage.sandpiles.sandpile
sage.dynamics.arithmetic_dynamics.wehlerK3
sage.dynamics.arithmetic_dynamics.projective_ds
sage.dynamics.arithmetic_dynamics.berkovich_ds
sage.dynamics.arithmetic_dynamics.product_projective_ds
sage.dynamics.arithmetic_dynamics.affine_ds
sage.dynamics.arithmetic_dynamics.projective_ds
normalize_coordinates()

N
nonsink_vertices() (sage.sandpiles.sandpile.Sandpile method), 71
nonspecial_divisors() (sage.sandpiles.sandpile.Sandpile method), 72
normal_form() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 195

O
nth_preimage_tree()
nth_iterate_psi()
nth_iterate_phi()
nth_iterate_map()
nth_iterate()
normalize_coordinates()
normalize_coordinates()

P
parallel_firing_graph() (in module sage.sandpiles.sandpile), 105

Index
Discrete dynamics, Release 10.1

partition_sandpile() (in module sage.sandpiles.sandpile), 105

periodic_points() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 163

PeriodicSolitonCellularAutomata (class in sage.dynamics.cellular_automata.solitons), 12

phi() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 225

picard_representatives() (sage.sandpiles.sandpile.Sandpile method), 72

plot() (sage.dynamics.cellular_automata.elementary.ElementaryCellularAutomata method), 7

plot() (sage.dynamics.cellular_automata.glca.GraftalLaceCellularAutomata method), 11

points() (sage.sandpiles.sandpile.Sandpile method), 73

polytope() (sage.sandpiles.sandpile.Sandpile method), 96

polytope_integer_pts() (sage.sandpiles.sandpile.Sandpile method), 96

possible_periods() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 166

possible_periods() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_finite_field method), 204

postcritical_set() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 167

postulation() (sage.sandpiles.sandpile.Sandpile method), 73

potential_good_reduction() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 196

preperiodic_points() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 168

primes_of_bad_reduction() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 171

print_state() (sage.dynamics.cellular_automata.elementary.ElementaryCellularAutomata method), 7

print_state() (sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata method), 24

print_state_evolution() (sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata method), 24

print_states() (sage.dynamics.cellular_automata.elementary.ElementaryCellularAutomata method), 8

print_states() (sage.dynamics.cellular_automata.glca.GraftalLaceCellularAutomata method), 11

print_states() (sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata method), 25

psi() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 225

Q

q_reduced() (sage.sandpiles.sandpile.SandpileDivisor method), 97

Qxa() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 211

Qyb() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 211

R

Ramification_poly() (sage.dynamics.arithmetic_dynamics.wehlerK3.WehlerK3Surface_ring method), 172

ramification_type() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 211

random_WehlerK3Surface() (in module sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 222

rank() (sage.sandpiles.sandpile.Sandpile method), 97

rational_preperiodic_graph() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 198

recurrents() (sage.sandpiles.sandpile.Sandpile method), 73

reduction_base_field() (sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine method), 125

reduce_base_field() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective_finite_field method), 173

reduced_form() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 199

reduced_laplacian() (sage.sandpiles.sandpile.Sandpile method), 74

reorder_vertices() (sage.sandpiles.sandpile.Sandpile method), 74

reset() (sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata method), 27

resolution() (sage.sandpiles.sandpile.Sandpile method), 75

resultant() (sage.dynamics.arithmetic_dynamics.berkovich_ds.DynamicalSystem_Berkovich method), 237

ring() (sage.sandpiles.sandpile.Sandpile method), 75

S

sage.dynamics.arithmetic_dynamics.affine_ds module, 116

sage.dynamics.arithmetic_dynamics.berkovich_ds module, 229
Index 249

sage.dynamics.arithmetic_dynamics.generic_ds module, 109
sage.dynamics.arithmetic_dynamics.product_projective_ds module, 204
sage.dynamics.arithmetic_dynamics.projective_ds module, 127
sage.dynamics.arithmetic_dynamics.wehlerK3 module, 208
sage.dynamics.cellular_automata.catalog module, 1
sage.dynamics.cellular_automata.elementary module, 1
sage.dynamics.cellular_automata.glca module, 9
sage.dynamics.cellular_automata.solitons module, 12
sage.dynamics.complex_dynamics.mandel_julia module, 31
sage.dynamics.finite_dynamical_system module, 39
sage.sandpiles.sandpile module, 53
sandlib() (in module sage.sandpiles.sandpile), 106
Sandpile (class in sage.sandpiles.sandpile), 58
sandpile() (sage.sandpiles.sandpile.SandpileConfig method), 87
sandpile() (sage.sandpiles.sandpile.SandpileDivisor method), 98
SandpileConfig (class in sage.sandpiles.sandpile), 81
SandpileDivisor (class in sage.sandpiles.sandpile), 89
scale_by() (sage.dynamics.arithmetic_dynamics.berkovich_ds.DynamicalSystem_Berkovich_projective method), 237
show() (sage.sandpiles.sandpile.Sandpile method), 76
show() (sage.sandpiles.sandpile.SandpileConfig method), 87
show() (sage.sandpiles.sandpile.SandpileDivisor method), 98
show3d() (sage.sandpiles.sandpile.Sandpile method), 76
sigma_invariants() (sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_projective method), 177
sigmaX() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeihlerK3Surface_ring method), 226
sigmaY() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeihlerK3Surface_ring method), 227
simulate_threshold() (sage.sandpiles.sandpile.SandpileDivisor method), 99
sink() (sage.sandpiles.sandpile.Sandpile method), 76
smith_form() (sage.sandpiles.sandpile.Sandpile method), 76
SolitonCellularAutomata (class in sage.dynamics.cellular_automata.solitons), 16
solve() (sage.sandpiles.sandpile.Sandpile method), 77
specialization() (sage.dynamics.arithmetic_dynamics.generic_ds.DynamicalSystem method), 115
stabilize() (sage.sandpiles.sandpile.SandpileConfig method), 88
stabilize() (sage.sandpiles.sandpile.SandpileDivisor method), 99
stable_configs() (sage.sandpiles.sandpile.Sandpile method), 77
state_evolution() (sage.dynamics.cellular_automata.solitons.SolitonCellularAutomata method), 28
stationary_density() (sage.sandpiles.sandpile.Sandpile method), 78
superstables() (sage.sandpiles.sandpile.Sandpile method), 78
support() (sage.sandpiles.sandpile.SandpileConfig method), 88
support() (sage.sandpiles.sandpile.SandpileDivisor method), 99
Sxa() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeihlerK3Surface_ring method), 212
Syb() (sage.dynamics.arithmetic_dynamics.wehlerK3.WeihlerK3Surface_ring method), 213
symmetric_recurrences() (sage.sandpiles.sandpile.Sandpile method), 79
T
triangle_sandpile() (in module sage.sandpiles.sandpile), 106
tutte_polynomial() (sage.sandpiles.sandpile.Sandpile method), 79
U
unsaturated_ideal() (sage.sandpiles.sandpile.Sandpile method), 80
unstable() (sage.sandpiles.sandpile.SandpileConfig method), 89
unstable() (sage.sandpiles.sandpile.SandpileDivisor method), 99
V
values() (sage.sandpiles.sandpile.SandpileConfig method), 89
values() (sage.sandpiles.sandpile.SandpileDivisor method), 100
verify_inverse_evolution() (sage.dynamics.finite_dynamical_system.InvertibleDiscreteDynamicalSystem method), 100
version() (sage.sandpiles.sandpile.Sandpile method), 80
W

WehlerK3Surface() (in module sage.dynamics.arithmetic_dynamics.wehlerK3), 208

WehlerK3Surface_field (class in sage.dynamics.arithmetic_dynamics.wehlerK3), 208

WehlerK3Surface_finite_field (class in sage.dynamics.arithmetic_dynamics.wehlerK3), 208

WehlerK3Surface_ring (class in sage.dynamics.arithmetic_dynamics.wehlerK3), 209

weierstrass_div() (sage.sandpiles.sandpile.SandpileDivisor method), 100

weierstrass_gap_seq() (sage.sandpiles.sandpile.SandpileDivisor method), 101

weierstrass_pts() (sage.sandpiles.sandpile.SandpileDivisor method), 101

weierstrass_rank_seq() (sage.sandpiles.sandpile.SandpileDivisor method), 102

weil_restriction() (sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine_field method), 126

wilmes_algorithm() (in module sage.sandpiles.sandpile), 106

Z

zero_config() (sage.sandpiles.sandpile.Sandpile method), 80

zero_div() (sage.sandpiles.sandpile.Sandpile method), 80