thompsons_v Documentation

Author

Aug 27, 2018

Contents

Nathan Barker, Andrew Duncan and David Robertson have submitted a paper entitled *The power conjugacy problem in Higman-Thompson groups* which addresses the following problem in the groups named $G_{n,r}$.

> *[AS74]* Given two elements x and y of a group, are there integers a and b and a third element z for which $x^a = zy^bz^{-1} \neq 1$? If so, provide them.

This package aims to implement the algorithms described in the paper. To do so it provides tools for working in

- the algebra $V_{n,r}$, a certain set of `words`, and
- the automorphism group $G_{n,r} = \mathrm{Aut}(V_{n,r})$.

This documentation serves three purposes. Firstly, it is meant to serve as a gentle introduction, explaining how to install and use the package. Secondly, it is meant to be a reference to all the various classes and functions provided by `thompson`. Finally, a number of examples are included throughout the documentation, which can be used as a means to test the implementation.

> David M. Robertson, Aug 27, 2018

CHAPTER 1

Overview

Warning: For the most part, this documentation is automatically generated by Sphinx , so it's not the prettiest thing that's ever been written.

This all began as a series of tools to draw tree pair diagrams in Thompson's group $V = G_{2,1}$, hence the name `thompson`. The focus has moved away from trees diagrams towards bijections between bases of words. From there, the package has grown and it now serves as an implementation of the algorithms we describe in the paper. Chief among these are the algorithms which:

- Determine the *type of component which contains a given element*, with respect to some given basis [Lemma 4.28]
- Computes the *quasi-normal basis* for a given automorphism [Lemma 4.28]
- Test to see if two words *share an orbit* [Lemma 4.34]
- Test to see if two automorphism are *conjugate* [Algorithm 5.6] and *power conjugate* [Algorithm 6.13].

CHAPTER 2

Implementation details

The implementation is written in Python and runs under Python 3.3 and above. The intent was to provide a proof-of-concept rather than a perfectly optimised implementation. Despite this, we have found the program useful as a calculator for $G_{n,r}$, as a means to generate examples, and for experimentally testing conjectures. The source code (both to the program and this documentation) is publicly available from GitHub.

Todo: Make this available under an open-source license.

CHAPTER 3

Contents

3.1 Getting Started

3.1.1 Installation

1. **Python interpreter.** This program is written in Python and needs a Python interpreter to run; to be specific, **Python 3.3 or above** is required. If you don't have this already installed on your system, your best bet is to download the latest version (currently 3.5) from the Python website. For Linux users, Python may be available from your distribution's repositories; take care to install version 3.x rather than 2.x!

2. **Source code.** If you have git available on your system, the easiest way to obtain the program is to simply clone the repository:

   ```
   git clone https://github.com/DMRobertson/thompsons_v.git
   ```

   If this is not possible, a ZIPped version can be downloaded from the project page on GitHub.

3. **Package requirements.** The de-facto package management tool for Python is the `pip` program, which is installed as part of Python 3.4 and higher. Python 3.3 users should consult the pip documentation for instructions on how to install `pip` directly. Some Linux distributions may also provide pip as a package (make sure it's for Python 3—it may be listed as `pip3`).

   Once pip is installed, navigate to the project directory and run

   ```
   pip install -r requirements.txt
   ```

   (possibly you may need to use `pip3`). This should do all the hard work for you.

   Direct installation: If you would prefer not to install `pip`, you may install the prerequisites directly. There is only one major Python package required to run the program: NetworkX, a library for working with graphs in Python. This may be available in Linux repositories, e.g. Ubuntu.

   This documentation is built using Sphinx. If you wish to build it yourself, or run the test suite, you must install Sphinx also. Again, consider checking Linux repositories e.g. in Ubuntu.

3.1.2 First steps

Navigate to the root folder (`thompsons_v`) in a terminal and start up Python using the `python` command. Once the python prompt (`>>>`) is available, try to `import thompson`:

```
$ python
Python 3.3.3 (v3.3.3:c3896275c0f6, Nov 18 2013, 21:19:30) [MSC v.1600 64 bit (AMD64)]␣
↪on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import thompson
>>>
```

If this fails, check that you're using Python 3.3 or above. If `python` started up Python 2.x, try using the command `python3` instead.

We can now use the python REPL as a sort of calculator for these groups.

```
>>> from thompson import *
>>> sig = (2, 1) #signature of the algebra we work in
>>> domain = Generators.standard_basis(sig).expand(0).expand(0)
>>> range  = Generators.standard_basis(sig).expand(0).expand(1)
>>> print(domain)
[x1 a1 a1, x1 a1 a2, x1 a2]
>>> print(range)
[x1 a1, x1 a2 a1, x1 a2 a2]
>>> phi = Automorphism(domain, range)
>>> print(phi)
InfiniteAut: V(2, 1) -> V(2, 1) specified by 3 generators (after expansion and␣
↪reduction).
x1 a1 a1 -> x1 a1
x1 a1 a2 -> x1 a2 a1
x1 a2    -> x1 a2 a2
```

```
>>> phi.image('x a2 a1 a2 a2 a1')
Word('x1 a2 a2 a1 a2 a2 a1', (2, 1))
>>> phi.order
inf
>>> phi.print_characteristics()
(-1, a1)
(1, a2)
>>> phi.dump_QNB()
x1 a1 Left semi-infinite component with characteristic (-1, a1)
x1 a2 Right semi-infinite component with characteristic (1, a2)
```

```
>>> print(phi ** 2)
InfiniteAut: V(2, 1) -> V(2, 1) specified by 4 generators (after expansion and␣
↪reduction).
x1 a1 a1 a1 -> x1 a1
x1 a1 a1 a2 -> x1 a2 a1
x1 a1 a2    -> x1 a2 a2 a1
x1 a2       -> x1 a2 a2 a2
>>> print(~phi) #inverse
InfiniteAut: V(2, 1) -> V(2, 1) specified by 3 generators (after expansion and␣
↪reduction).
x1 a1    -> x1 a1 a1
x1 a2 a1 -> x1 a1 a2
x1 a2 a2 -> x1 a2
```

(continues on next page)

(continued from previous page)

```
>>> print(~phi * phi) # == identity
PeriodicAut: V(2, 1) -> V(2, 1) specified by 1 generators (after expansion and␣
↪reduction).
x1 -> x1
```

3.1.3 Locating examples

A *number of examples* are included in this package, some of which are used in *[BDR]* . To access them, use the *load_example()* function:

```
>>> from thompson import *
>>> phi = load_example('nathan_pond_example')
>>> print(phi)
InfiniteAut: V(2, 1) -> V(2, 1) specified by 7 generators (after expansion and␣
↪reduction).
x1 a1 a1 a1 a1 -> x1 a1 a1
x1 a1 a1 a1 a2 -> x1 a1 a2 a1 a1
x1 a1 a1 a2 a1 -> x1 a2 a2
x1 a1 a1 a2 a2 -> x1 a2 a1
x1 a1 a2       -> x1 a1 a2 a1 a2
x1 a2 a1       -> x1 a1 a2 a2 a2
x1 a2 a2       -> x1 a1 a2 a2 a1
```

Note: Previously, one would access this example by using `from thompson.examples import nathan_pond_example`. I changed this, because it meant that every example was loaded (and the quasinormal bases etc. computed) whenever `thompson.examples` was imported.

To see the list of available examples, consult the *documentation for the examples module*. You might also like to consult the demo notebook on GitHub.

3.1.4 Running the test suite

Make sure Sphinx is installed (see the installation section). Then navigate to the `docs` folder and run `make doctest`. This should look through the source code for short tests and alert you if any of them fail. You can also run `make html` to build this documentation. Reset with `make clean`.

3.2 Number theory

At the bottom level of the package hierarchy are various helper functions which implement standard number-theoretic algorithms.

`thompson.number_theory.`**`gcd`**(*a*, *b*)

Calculate the Greatest Common Divisor of a and b.

Unless b==0, the result will have the same sign as b (so that when b is divided by it, the result comes out positive).

```
>>> gcd(12, 8)
4
```

(continues on next page)

(continued from previous page)

```
>>> gcd(0, 50)
50
>>> gcd(7, 101)
1
```

thompson.number_theory.**extended_gcd**(*a*, *b*)

From this exposition of the extended gcd algorithm. Computes $d = \gcd(a, b)$ and returns a triple (d, x, y) where $d = ax + by$.

```
>>> extended_gcd(14, 33)
(1, -7, 3)
>>> extended_gcd(12, 32)
(4, 3, -1)
```

thompson.number_theory.**lcm**(**args*)

Computes the least common multiple of the given *args*. If a single argument is provided, the least common multiple of its elements is computed.

```
>>> n = randint(1, 1000000)
>>> lcm(n) == n
True
>>> lcm(2, 13)
26
>>> lcm(*range(1, 10)) #1, 2, ..., 9
2520
>>> lcm(range(1, 10))  #Don't have to unpack the arguments
2520
```

thompson.number_theory.**solve_linear_diophantine**(*a*, *b*, *c*)

Solves the equation $ax + by = c$ for integers $x, y \in \mathbb{Z}$.

Return type

None if no solution exists; otherwise a triple *(base, inc, lcm)* where

- *base* is a single solution (x_0, y_0);
- *inc* is a pair $(\delta x, \delta y)$ such that if (x, y) is a solution, so is $(x, y) \pm (\delta x, \delta y)$; and
- *lcm* is the value of $\mathrm{lcm}(a, b)$.

thompson.number_theory.**solve_linear_congruence**(*a*, *b*, *n*)

Solves the congruence $ax \equiv b \pmod{n}$.

Return type a *SolutionSet* representing all such solutions x.

```
>>> x = solve_linear_congruence(6, 12, 18)
>>> print(x)
{..., -1, 2, 5, 8, 11, 14, ...}
>>> y = solve_linear_congruence(5, 7, 11)
>>> print(y)
{..., -3, 8, 19, 30, 41, 52, ...}
>>> print(x & y)
{..., -25, 8, 41, 74, 107, 140, ...}
```

thompson.number_theory.**divisors**(*n*, *include_one=True*)

An iterator that yields the positive divisors $d \mid n$.

Parameters **include_one** (*bool*) – set to False to exlude $d = 1$.

```
>>> list(divisors(12))
[1, 2, 3, 4, 6, 12]
>>> list(divisors(125, include_one=False))
[5, 25, 125]
```

thompson.number_theory.**prod**(*iterable*)

Handy function for computing the product of an iterable collection of numbers. From Stack Overflow.

```
>>> prod(range(1, 5))
24
```

3.3 Words and standard forms

By a *word*, we mean a string written using the symbols

$$X \cup \Omega = \{x_1, \ldots, x_r\} \cup \{\alpha_1, \ldots, \alpha_n, \lambda\}.$$

We treat the x_i are constants, the α_i are unary operators and λ as an *n*-ary operation. We refer to the parameters n and r as the *arity* and *alphabet size* respectively.

3.3.1 Notation

If we collect together all such words, we obtain an *algebra* (in the sense of universal algebra) of words. We consider three different algebras; using the notation of *[Cohn81]*:

1. $W(\Omega; X)$, the set of all finite strings written using letters in $X \cup \Omega$. In other words/jargon, this is the free monoid on $X \cup \Omega$. Cohn calls these strings Ω-rows.
2. $W_\Omega(X)$, the subset of $W(\Omega; X)$ whose strings represent a `valid` series of operations. *Valid* means that each the operations α_i and λ always recieve the correct number of arguments. Cohn calls these strings Ω-words.
3. $V_{n,r}(X) = W_\Omega(X)/\mathfrak{q}$. This is equivalent to the set of words in $W_\Omega(X)$ which are in Higman's `standard form`. Roughly speaking, a word is in standard form if is valid (type 2) and it cannot be reduced to a shorter word using the following two rules.
   (a) $w_1 \ldots w_n \lambda \alpha_i = w_i$, where each w_i is in standard form.
   (b) $w\alpha_1 \ldots w\alpha_n \lambda = w$, where w is in standard form.

Sometimes we need to refer to a string which consists only of α-s. Write A for the set $\{\alpha_1, \ldots, \alpha_n\}$. We define A^* to be the set of finite strings over A. (Again, this is the free monoid on A.)

Finally, let S be a set of words. We define $X\langle A\rangle$ to be the set of concatenations $s\Gamma$, where $s \in S$ and $\Gamma \in A^*$. It is sometimes helpful to think of $S\langle A\rangle$ as the set of words *below* S.

See also:

Sections 2 and 2.3 and Remark 3.3 of the paper.

3.3.2 Signatures

It may happen that we need to work in different algebras $V_{n,r}, V_{m,s}$ at the same time. To keep track of the algebras that different words belong to, we associate a `Signature` to each word $w \in V_{n,r}$.[1] Signatures are implemented as a glorified tuple (n, r). This means they are immutable.

[1] This made it easier to catch bugs before they happened when I was passing words around as arguments to functions.

class thompson.word.**Signature**

static __new__(*cls*, *arity*, *alphabet_size*)
Signatures store an *arity* $n \geq 2$ and an *alphabet_size* $r \geq 1$.

__contains__(*other*)
We can use the `in` operator to see if a *Word* lies in the algebra that this signature describes.

```
>>> s = Signature(2, 1)
>>> Word('x1 a1 a2', s) in s
True
>>> Word('x1 a1 a2', (2, 2)) in s
False
```

is_isomorphic_to(*other*)
We can test the (signatures of) two algebras $V_{n,r}$ and $V_{m,s}$ to see if they are isomorphic. This happens precisely when $n = m$ and $r \equiv s \pmod{n-1}$.

```
>>> Signature(2, 1).is_isomorphic_to(Signature(2, 4))
True
>>> Signature(2, 1).is_isomorphic_to(Signature(3, 1))
False
>>> Signature(3, 2).is_isomorphic_to(Signature(3, 1))
False
>>> Signature(3, 3).is_isomorphic_to(Signature(3, 1))
True
```

See also:

Corollary 3.14 of the paper.

3.3.3 Operations on words

We represent a word as a `tuple` of integers, where:

- x_i is represented by i,
- α_i is represented by $-i$, and
- λ is represented by 0.

We can write words of all types in this format, but we're only interested in the standard forms (*type 3*). To this end, the *validate()* detects those which are of *type 2*, and *standardise()* reduces type 2 words to type 3.

thompson.word.**format**(*word*)
Turns a *Word*, or a sequence of integers representing a *word* into a nicely formatted string. Can be thought of as an inverse to *from_string()*. The *word* is not processed or reduced in any way by this function.

```
>>> format(Word("x1 a1 a2 a1 a1", (2, 1)))
'x1 a1 a2 a1 a1'
>>> format([2, -1, 2, -2, 0])
'x2 a1 x2 a2 L'
>>> format([])
'<the empty word>'
```

thompson.word.**format_cantor**(*word*, *subset=False*)
An alternative, more concise notation for simple words. We use the Cantor set notation, where we write $n-1$ in place of *lpha_n*. All other symbols λ and x_i are omitted, so this should really only be used on simple words.

The *subset* argument changes how we display a single x_i. It's here for experimental use by *CantorSubset.__str__*.

```
>>> format_cantor(Word("x1 a1 a2 a1 a1", (2, 1)))
'0100'
>>> format_cantor([2, -1, 2, -2, 0])
'01'
>>> format_cantor([])
'<the empty word>'
>>> format_cantor([1])
'<root>'
>>> format_cantor([1], subset=True)
'<entire Cantor set>'
>>> format_cantor([-1])
'0'
```

thompson.word.**from_string**(*string*)

Converts a *string* representing a word to the internal format—a tuple of integers. No kind of validation or processing is performed. This is used internally by other functions and doesn't typically need to be called directly. Two forms of string are accepted: 'Higman' and 'Cantor'. We assume the former is in use if the string contains an `x`, `a` or `L`; else we assume the latter. The first is the Higman notation (involving x's, : $math$:alpha's and λ's).$Anythingwhichdoesnotdenoteabasisletter(e.g.$'x', ``'x2', 'x45') a descendant operator (e.g. `'a1'`, `'a3'`, `'a27'`) or a contraction (`'L'` for λ) is ignored. Note that `'x'` is interpreted as shorthand for `'x1'`.

```
>>> x = from_string('x2 a1 x2 a2 L')
>>> print(x, format(x), sep='\n')
(2, -1, 2, -2, 0)
x2 a1 x2 a2 L
>>> x = from_string('x a1 a2 a3')
>>> print(x, format(x), sep='\n')
(1, -1, -2, -3)
x1 a1 a2 a3
>>> w = random_simple_word()
>>> from_string(str(w)) == w
True
```

The second form is the Cantor-set notation. This only applies when there is a single basis letter x, which is not written down as part of the format. We use this mode if the first non-whitespace character of the input *string* is not `x`. A descendant operator α_n is written as the integer $n-1$. All characters outside the range 0–9 are ignored.

```
>>> x = from_string("001101")
>>> print(x, format(x), format_cantor(x), sep='\n')
(1, -1, -1, -2, -2, -1, -2)
x1 a1 a1 a2 a2 a1 a2
001101
```

Return type `tuple` of integers

Todo: More convenient ways of inputting words, e.g. `x a1^3` instead of `x a1 a1 a1`. Use of unicode characters α and λ?

---

thompson.word.**validate**(*letters*, *signature*)

Checks that the given *letters* describe a valid word belonging to the algebra with the given *signature* = *(arity, alphabet_size)*. If no errors are raised when running this function, then we know that:

- The first letter is an x_i.
- Every λ has *arity* arguments to its left.
- If the word contains a λ, every subword is an argument of some λ.
- No α_i appears with $i >$ *arity*.
- No x_i occurs with $i >$ *alphabet_size*.

Calling this function does not modify *letters*. The argument *letters* need not be in standard form.

```
>>> w = from_string('a1 x a2 L'); w
(-1, 1, -2, 0)
>>> validate(w, (2, 1))
Traceback (most recent call last):
        ...
ValueError: The first letter (a1) should be an x.
>>> validate(from_string('x1 a2 x12 a1 a2 L'), (2, 4))
Traceback (most recent call last):
        ...
IndexError: Letter x12 at index 2 is not in the alphabet (maximum is x4).
>>> validate(from_string('x1 a1 a2 a3 a4 a5 x2 a2 L'), (3, 2))
Traceback (most recent call last):
        ...
IndexError: Letter a4 at index 4 is not in the alphabet (maximum is a3).
>>> validate(from_string('x1 a2 L'), (4, 1))
Traceback (most recent call last):
        ...
ValueError: Letter L at index 2 is invalid. Check that lambda has 4 arguments.
>>> validate(from_string('x1 x1 L x1 L x1'), (2, 1))
Traceback (most recent call last):
        ...
ValueError: Word is invalid: valency is 2 (should be 1).
```

Raises

- **ValueError** – if the first letter is not an x_i.
- **IndexError** – if this word contains an x_i with i outside of the range 1 ... *alphabet_size*
- **IndexError** – if this word contains an α_i outside of the range 1 ... *arity*
- **IndexError** – if this word is empty (i.e. consists of 0 letters).
- **ValueError** – if this word fails the valency test.

See also:

Proposition 2.12 of the paper for the *valency test*.

`thompson.word.`**`standardise`**(*letters*, *signature*, *tail=()*)

Accepts a valid word as a `tuple` of *letters* and reduces it to standard form. The result—a new tuple of letters—is returned. The *signature* must be given so we know how many arguments to pass to each λ. The *tail* argument is used internally in recursive calls to this function and should be omitted.

In the examples below, the *`Word`* class standardises its input before creating a Word.

```
>>> #Extraction only
>>> print(Word("x a1 a2 a1 x a2 a1 L a1 a2", (2, 1)))
x1 a1 a2 a1 a2
>>> #Contraction only
```

(continues on next page)

(continued from previous page)

```
>>> print(Word("x2 a2 a2 a1 x2 a2 a2 a2 L", (2, 2)))
x2 a2 a2
>>> #Contraction only, arity 3
>>> print(Word("x1 a1 a1 x1 a1 a2 x1 a1 a3 L", (3, 2)))
x1 a1
>>> #Both extraction and contraction
>>> print(Word("x a1 a2 a1 a1 x a1 a2 a1 x a2 a1 L a1 a2 a1 x a1 a2 a1 a2 a2 L L",
↪ (2, 1)))
x1 a1 a2 a1
>>> #Lambda concatenation
>>> print(Word("x a1 a2 a1 x a1 a2 a1 x a2 a1 L a1 a2 a1 x a1 a2 a1 a2 a2 L L",␣
↪(2, 1)))
x1 a1 a2 a1 x1 a1 a2 a1 a2 L
>>> #A mix of contraction and extraction
>>> print(Word("x a2 x a1 L x a1 L a1 a2 a1 a2", (2, 1)))
x1 a1 a1 a2
>>> #Can't contract different x-es
>>> print(Word('x1 a1 x2 a2 L', (2, 2)))
x1 a1 x2 a2 L
>>> #Something meaty, arity 3
>>> print(Word('x1 a1 x1 a2 x1 a3 L a2 a1 a3 x1 a2 a1 x2 a1 x1 a1 a1 x1 a1 a2 x2␣
↪L x1 a2 L a2 a1 a1 L a3 a3 a2', (3, 2)))
x1 a1 a1 a1 a3 a2
```

Raises

- **TypeError** – if *letters* is not a tuple of integers.
- **IndexError** – if *letters* describes an *invalid* word.

Return type tuple of integers.

See also:

Remark 3.3 of the paper.

thompson.word.**lambda_arguments**(*word*, *signature=None*)

This function takes a *valid* word which ends in a λ, and returns the arguments of the rightmost λ as a list of *Words*.

If *word* is given as a *Word* instance, then *signature* may be omitted. Otherwise the *signature* should be provided.

```
>>> w = Word('x a1 a2 x a2 a2 L x a1 a1 L', (2, 1))
>>> subwords = lambda_arguments(w)
>>> for subword in subwords: print(subword)
x1 a1 a2 x1 a2 a2 L
x1 a1 a1
>>> w = 'x x x x a1 x L x a1 x a2 x L L x a1 a2 x L x a1 a2 x a2 a1 x L L'
>>> subwords = lambda_arguments(Word(w, (3, 1)))
>>> for subword in subwords: print(subword)
x1
x1 x1 x1 a1 x1 L x1 a1 x1 a2 x1 L L x1 a1 a2 x1 L
x1 a1 a2 x1 a2 a1 x1 L
```

Raises

- **IndexError** – if *word* is an empty list of letters.
- **ValueError** – if the last letter in *word* is not a lambda.

- **TypeError** – if no arity is provided and *word* has no arity attribute.

Return type list of *Words*.

thompson.word.**are_contractible**(*words*)

Let *words* be a list of words, either as sequences of integers or as full *Word* objects. This function tests to see if *words* is a list of the form $(w\alpha_1, \ldots, w\alpha_n)$, where `n == len(words)`.

Returns w (as a `tuple` of integers) if the test passes; the empty tuple if the test fails.

Warning: If there is a prefix w, this function does **not** check to see if

- w *is a valid word*, or
- n is the same as the arity of the context we're working in.

```
>>> prefix = are_contractible(
...     [Word("x a1 a2 a3 a1", (3, 1)), Word("x a1 a2 a3 a2", (3, 1)), Word("x a1␣
↪a2 a3 a3", (3, 1))])
>>> format(prefix)
'x1 a1 a2 a3'
```

thompson.word.**free_monoid_on_alphas**(*arity*)

An infinite iterator which enumerates the elements of A^* in shortlex order.

```
>>> for i, gamma in enumerate(free_monoid_on_alphas(4)):
...     if i >= 10: break
...     print(format(gamma))
<the empty word>
a1
a2
a3
a4
a1 a1
a1 a2
a1 a3
a1 a4
a2 a1
```

thompson.word.**root**(*sequence*)

Given a sequence $\Gamma \in A^*$, this function computes the root $\sqrt{\Gamma} \in A^*$ and the root power $r \in \mathbb{N}$. These are objects such that $(\sqrt{\Gamma})^r = \Gamma$, and r is maximal with this property.

Returns the pair $(\sqrt{\Gamma}, r)$

```
>>> root('abababab')
('ab', 4)
>>> root([1, 2, 3, 1, 2, 3])
([1, 2, 3], 2)
```

See also:

The discussion following Corollary 6.7 in the paper.

3.3.4 The Word class

It's important to know when a sequence of letters denotes a word in standard form. The *Word* class addresses this problem by standardising its input upon creation. We can think of a Word object as a fixed list of letters with the guarantee that it is in standard form. Words are also given a *Signature* at creation time, so that we know which algebra the word comes from.

We also need to know that once a Word has been created, it cannot be modified to something not in standard form. We achieve this simply by making it impossible to modify a Word. Words are implemented as (subclasses of) `tuple`, which are immutable. One useful side effect of this is that Words can be used as dictionary keys.[2]

```
>>> w = Word('x1 a1', (2, 1))
>>> x = {}; x[w] = 'stored value'
>>> x[w]
'stored value'
>>> #Can also use the underlying tuple as a key
>>> x[1, -1]
'stored value'
>>> #the reason this works
>>> hash(w) == hash((1, -1))
True
```

class thompson.word.**Word**

Variables

- **signature** – the signature of the algebra this word belongs to.
- **lambda_length** – the number of times the contraction symbol λ occurs in this word after it has been written in standard form.

static __new__(*cls*, *letters*, *signature*, *preprocess=True*)

Creates a new Word consisting of the given *letters* belonging the algebra with the given *signature*. The *letters* may be given as a list of integers or as a string (which is passed to the *from_string()* function). The signature can be given as a tuple or a fully-fledged *Signature*.

```
>>> Word("x2 a1 a2 a3 x1 a1 a2 a1 x2 L a2", (3, 2))
Word('x1 a1 a2 a1', (3, 2))
```

By default, the argument *preprocess* is True. This means that words are *validated* and *reduced to standard form* before they are stored as a tuple. These steps can be disabled by using `preprocess=False`. This option is used internally when we *know* we have letters which are already vaild and in standard form.

Raises **ValueError** – See the errors raised by *validate()*.

address(*include_root=False*)

A shorthand to *format_cantor()* for printing *simple* words. By default, it discards the the root x_i.

```
>>> w = Word("x2 a1 a3 a1 a1 a2", (3, 4))
>>> w.address()
'02001'
>>> Word("x2 a1 a3 a1 a1 a2", (3, 4)).address(include_root=True)
'x202001'
```

__lt__(*other*)

[2] This made it so much easier to cache the images of *homomorphisms*, because I could just use a vanilla Python dictionary.

Let us assign a total order to the alphabet $X \cup \Omega$ by declaring:

$$\alpha_1 < \cdots < \alpha_n < x_1 < \cdots < x_r < \lambda$$

We can use this to order *simple words* by using dictionary order.

```
>>> Word('x1', (2, 2)) < Word('x1', (2, 2))
False
>>> Word('x1', (2, 2)) < Word('x2', (2, 2))
True
>>> Word('x1 a2', (2, 2)) < Word('x1 a1', (2, 2))
False
>>> Word('x1 a1 a2 a3 a4', (4, 1)) < Word('x1 a1 a2 a3 a3', (4, 1))
False
```

We extend this to non-simple words *in standard form* in the following way. Let $\lambda(u)$ denote the lambda-length of u.

1. If $\lambda(u) < \lambda(v)$, then $u < v$.

```
>>> Word('x2 x1 L', (2, 2)) < Word('x1 x2 x1 L L', (2, 2))
True
```

2. If $u = v$, then neither is strictly less than the other.

```
>>> Word('x1 x2 L', (2, 2)) < Word('x1 x2 L', (2, 2))
False
```

3. Otherwise $u \neq v$ are different words with the same λ-length. Break both words into the n arguments of the outmost lambda, so that $u = u_1 \dots u_n \lambda$ and $v = v_1 \dots v_n \lambda$, where each subword is in standard form.
4. Let i be the first index for which $u_i \neq v_i$. Test if $u_i < v_i$ by applying this definition recursively. If this is the case, then $u < v$; otherwise, $u > v$.

```
>>> #True, because x2 < x2 a2
>>> Word('x1 x2 L', (2, 2)) < Word('x1 x2 a2 L', (2, 2))
True
>>> #True, because words of lambda-length 1 are less than those of lambda-
↪length 2.
>>> Word('x1 x2 L x3 x4 L L', (2, 4)) < Word('x1 x2 L x3 L x4 L', (2, 4))
True
```

The other three comparison operators (<=, >, >=) are also implemented.

```
>>> Word('x1 x2 L', (2, 2)) <= Word('x1 x2 L', (2, 2))
True
>>> Word('x1 a1', (2, 2)) <= Word('x1 a1', (2, 2)) <= Word('x1 a1 a2', (2, 2))
True
>>> Word('x1', (2, 2)) > Word('x1', (2, 2))
False
>>> Word('x1', (2, 2)) >= Word('x1', (2, 2))
True
>>> Word('x1 a2', (2, 2)) > Word('x1', (2, 2))
True
>>> Word('x1 a2', (2, 2)) >= Word('x1', (2, 2))
True
```

Todo: I think this is a total order—try to prove this. I based the idea on *[Zaks]* (section 2, definition 1) which describes a total order on k-ary trees. On another note, isn't this similar to shortlex order?

is_simple()

Let us call a Word *simple* if it has a lambda length of zero once it has been reduced to standard form.

```
>>> Word("x1 a1 a2", (2, 1)).is_simple()
True
>>> Word("x1 a1 a2 x2 a2 L", (2, 2)).is_simple()
False
>>> #Simplify a contraction
>>> Word("x1 a1 a1 x1 a1 a2 L", (2, 1)).is_simple()
True
>>> #Lambda-alpha extraction
>>> Word("x1 x2 L a2", (2, 2)).is_simple()
True
```

is_above(*word*)

Tests to see if the current word is an initial segment of *word*. Returns True if the test passes and False if the test fails.

```
>>> w = Word('x1 a1', (2, 2))
>>> w.is_above(Word('x1 a1 a1 a2', (2, 2)))
True
>>> w.is_above(Word('x1', (2, 2)))
False
>>> w.is_above(w)
True
>>> v = Word('x1 a1 x1 a2 L', (2, 2))
>>> w.is_above(v)
False
>>> v.is_above(w)
True
```

is_above_set(*generators*)

test_above(*word*)

Tests to see if the current word c is an initial segment of the given *word* w. In symbols, we're testing if $w = c\Gamma$, where $\Gamma \in \langle A \rangle$ is some string of alphas only.

The test returns Γ (as a tuple of integers) if the test passes; note that Γ could be the empty word 1 (if $c = w$). If the test fails, returns `None`.

```
>>> c = Word('x1 a1', (2, 2))
>>> c.test_above(Word('x1 a1 a1 a2', (2, 2)))
(-1, -2)
>>> c.test_above(Word('x1', (2, 2))) is None
True
>>> c.test_above(c)
()
>>> w = Word('x1 a2 x1 a1 L', (2, 2))
>>> print(c.test_above(w))
None
>>> w.test_above(c)
(-2,)
>>> v = Word('x1 a2 x1 a1 L x1 a2 a2 L x1 a2 x1 a1 L L', (2, 2))
```

(continues on next page)

(continued from previous page)

```
>>> print(c.test_above(v))
None
>>> #There are two possible \Gamma values here; only one is returned.
>>> v.test_above(c)
(-1, -1, -2)
```

Note: There may be many different values of Γ for which $w = c\Gamma$; this method simply returns the first such Γ that it finds.

max_depth_to(*basis*)

Let w be the current word and let X be a *basis*. Choose a (possibly empty) string of alphas Γ of length $s \geq 0$ at random. What is the smallest value of s for which we can guarantee that $w\Gamma$ is below X, i.e. in $X\langle A \rangle$?

```
>>> from thompson.generators import Generators
>>> basis = Generators.standard_basis((2, 1)).expand(0).expand(0).expand(0)
>>> basis
Generators((2, 1), ['x1 a1 a1 a1', 'x1 a1 a1 a2', 'x1 a1 a2', 'x1 a2'])
>>> Word('x', (2, 1)).max_depth_to(basis)
3
>>> Word('x a1', (2, 1)).max_depth_to(basis)
2
>>> Word('x a2 x a1 L x1 a1 L', (2, 1)).max_depth_to(basis)
4
>>> Word('x a2', (2, 1)).max_depth_to(basis)
0
```

Warning: If *basis* doesn't actually generate the algebra we're working in, this method could potentially loop forever.

as_interval()

Returns a pair *(start, end)* of `Fractions` which describe the interval $I \subseteq [0, 1]$ that this word corresponds to.

```
>>> def print_interval(w, s):
...     start, end = Word(w, s).as_interval()
...     print('[{}, {}]'.format(start, end))
...
>>> print_interval('x a1 a2 a1 x L', (2, 1))
Traceback (most recent call last):
...
ValueError: The non-simple word x1 a1 a2 a1 x1 L does not correspond to a␣
↪interval.
>>> print_interval('x1', (2, 1))
[0, 1]
>>> print_interval('x1', (3, 1))
[0, 1]
>>> print_interval('x1', (4, 2))
[0, 1/2]
>>> print_interval('x1 a1', (2, 1))
[0, 1/2]
>>> print_interval('x1 a1', (3, 1))
```

(continues on next page)

(continued from previous page)

```
[0, 1/3]
>>> print_interval('x1 a1', (4, 2))
[0, 1/8]
>>> print_interval('x1 a2 a1 a2', (2, 1))
[5/8, 3/4]
>>> print_interval('x1 a2 a1 a2', (3, 1))
[10/27, 11/27]
>>> print_interval('x1 a2 a1 a2', (4, 2))
[17/128, 9/64]
```

```
>>> from thompson.examples import random_simple_word
>>> s, e = random_simple_word().as_interval(); 0 <= s < e <= 1
True
```

Raises **ValueError** – if the word *is not simple*.

static ray_as_rational(*base*, *spine*)

Converts the infinite string base spine$^\infty$ to a rational in the unit interval.

```
>>> Word.ray_as_rational(Word("x", (2,1)), from_string("a1 a2"))
Fraction(1, 3)
>>> Word.ray_as_rational(Word("x a1", (2,1)), from_string("a2 a1"))
Fraction(1, 3)
```

alpha(*index*)

Let w stand for the current word. This method creates and returns a new word $w\alpha_{\text{index}}$.

```
>>> Word("x a1 a2", (3, 2)).alpha(3)
Word('x1 a1 a2 a3', (3, 2))
```

Raises **IndexError** – if *index* is not in the range 1... *arity*.

extend(*tail*)

Concatenates the current word with the series of letters *tail* to form a new word.The argument *tail* can be given as either a string or a tuple of integers.

```
>>> Word('x1 a2 a1', (2, 1)).extend('a2 a2')
Word('x1 a2 a1 a2 a2', (2, 1))
>>> Word('x1 a2 a1', (2, 1)).extend('x1 a2 a2')
Traceback (most recent call last):
        ...
ValueError: Word is invalid: valency is 2 (should be 1).
>>> Word('x1 a2 a1', (2, 1)).extend('x1 a2 a2 L')
Word('x1 a2', (2, 1))
```

expand()

Returns an iterator that yields the *arity* descendants of this word.

```
>>> w = Word("x a1", (5, 1))
>>> for child in w.expand():
...     print(child)
x1 a1 a1
x1 a1 a2
x1 a1 a3
```

(continues on next page)

(continued from previous page)

```
x1 a1 a4
x1 a1 a5
>>> w = Word("x a1 a1 x a2 a1 L", (2, 1))
>>> for child in w.expand():
...     print(child)
x1 a1 a1
x1 a2 a1
```

Return type iterator which yields *Words*.

split(*head_width*)

Splits the current word *w* into a pair of tuples *head*, *tail* where `len(head) == head_width`. The segments of the word are returned as tuples of integers, i.e. not fully fledged *Words*.

Raises **IndexError** – if *head_width* is outside of the range `0 <= head_width <= len(w)`.

```
>>> Word('x1 a1 a2 a3 a1 a2', (3, 1)).split(4)
((1, -1, -2, -3), (-1, -2))
>>> Word('x1 a1 a2 a3 a1 a2', (3, 1)).split(10)
Traceback (most recent call last):
        ...
IndexError: The head width 10 is not in the range 0 to 6.
```

rsplit(*tail_width*)

The same as *split()*, but this time *tail_width* counts from the right hand side. This means that `len(tail) == tail_width`.

Raises **IndexError** – if *tail_width* is outside of the range `0 <= tail_width <= len(w)`.

```
>>> Word('x1 a1 a2 a3 a1 a2', (3, 1)).rsplit(4)
((1, -1), (-2, -3, -1, -2))
>>> Word('x1 a1 a2 a3 a1 a2', (3, 1)).rsplit(10)
Traceback (most recent call last):
        ...
IndexError: The tail width 10 is not in the range 0 to 6.
```

shift(*delta=1*, *signature=None*)

Takes a simple word $x_i\Gamma$ and returns the word $x_{i+\delta}\Gamma$. The target word has signature $(n, r+\delta)$ by default where (n, r) is the current word's signature. This can be overridden by passing in a *signature*.

```
>>> w = Word("x1 a1 a2 x1 a2 x2 L", (3, 2))
>>> w.shift()
Traceback (most recent call last):
...
ValueError: Can only shift simple words.
>>> v = Word("x1 a2 a3", (3, 1))
>>> v.shift()
Word('x2 a2 a3', (3, 2))
```

Raises

- **ValueError** – if *delta* is not a positive integer.
- **ValueError** – if the given word is not *simple*

subwords(*discard_root=False*)
An iterator method which yields the anscestors of a *simple word*. Use *discard_root* if you want to ignore words which correspond to x_i.

```
>>> w = Word("x1 a1 a2 a1 a2", (2, 1))
>>> print( *w.subwords(), sep="\n")
x1
x1 a1
x1 a1 a2
x1 a1 a2 a1
x1 a1 a2 a1 a2
>>> print( *w.subwords(discard_root=True), sep="\n" )
x1 a1
x1 a1 a2
x1 a1 a2 a1
x1 a1 a2 a1 a2
```

3.3.5 Next steps

Now that we can work with words in $V_{n,r}$, we need to be able to work with *collections* of words. The *generators* module will let us treat a list of words as a generating set and examine the subalgebra that the collection generates.

3.4 Generating sets and bases

Let S be a set of words in $V_{n,r}$. We can obtain new words by applying the operations α_i and λ to elements of S. Next, we can apply the operations to the new words we've received. This process continues indefinitely, producing a set of words T whose elements are sourced from the original set S. We call T the *subalgebra generated by* S and say that S *generates* T.

Of particular interest are *bases*, which we can think of as generating sets which do not contain any redundant elements. For example, it would be pointless to include each of x, $x\alpha_1$ and $x\alpha_2$ in S, as the first can be obtained from the last two and vice versa.

3.4.1 The Generators class

class thompson.generators.**Generators**(*signature*, *generators=None*)
An ordered subset of $V_{n,r}$, together with methods which treat such sets as generating sets and bases. Internally, this is a subclass of `list`, so it is possible to reorder and otherwise modify collections of Generators.

Variables **signature** – The *Signature* of the algebra this set belongs to.

__init__(*signature*, *generators=None*)
When creating a generating set, you must specify the algebra $V_{n,r}$ it is a subset of. A list of generators can optionally be provided. Each generator is passed to *append()*.

append(*w*)
Adds *w* to this generating set. If *w* is a string, a *Word* object is created and assigned the same *arity* and *alphabet_size* as the generating set.

Raises

- **TypeError** – if *w* is neither a string nor a Word.
- **IndexError** – if *w* has a different arity to the generating set.

- **IndexError** – if *w* has a larger alphabet_size the generating set.
- **ValueError** – if *w* is already contained in this generating set.

extend(*iterable*) → None – extend list by appending elements from the iterable

__eq__(*other*)

Two Generators instances are equal iff they have the same signature and the same elements in the same order.

```
>>> x = random_basis(signature=(3, 2))
>>> y = x.embed_in((4, 3))
>>> x == y #Different signatures, so False
False
>>> list.__eq__(x, y) #Even though they have the same elements
True
```

copy()

We override `list.copy()` so that we don't have to recreate *Generators* instances by hand.

```
>>> olga_f = load_example('olga_f')
>>> X = olga_f.quasinormal_basis.copy()
>>> X is olga_f.quasinormal_basis
False
>>> type(X).__name__
'Generators'
```

filter(*func*)

Creates a copy of the current generating set whose elements *x* are those for which *func(x)* is `True`. The original generating set is unmodified, and the order of elements is inherited by the filtered copy.

```
>>> X = load_example('olga_f').quasinormal_basis
>>> print(X)
[x1 a1 a1, x1 a1 a2, x1 a2 a1, x1 a2 a2 a1, x1 a2 a2 a2]
>>> def test(x):
...     return len(x) % 2 == 0
...
>>> print(X.filter(test))
[x1 a2 a2 a1, x1 a2 a2 a2]
```

test_free()

Tests to see if the current generating set X is a free generating set. To do so, the words contained in X must all be *simple*.

If the test fails, returns the first pair of indices (i, j) found for which X_i *is above* X_j. If the test passes, returns $(-1, -1)$.

```
>>> g = Generators((2, 3), ["x1 a1", "x1 a2 a1", "x1 a2 a1 a1", "x1 a2 a2"])
>>> g.test_free()
(1, 2)
>>> print(g[1], g[2], sep='\n')
x1 a2 a1
x1 a2 a1 a1
>>> g = Generators((2, 1), ['x1 a2 a2', 'x1 a2 a1 x1 a2 a2 a2 a1 L x1 a1 L'])
>>> g.test_free()
Traceback (most recent call last):
...
ValueError: Cannot test for freeness unless all words are simple.
```

Raises `ValueError` – if any word in the basis is not simple.

See also:

Lemma 3.16 of the paper.

is_free()
Returns True if this is a free generating set; otherwise False.

```
>>> g = Generators((2, 3), ['x1', 'x2']);
>>> g.is_free()
True
>>> g.append('x3'); g.is_free()
True
>>> g.append('x2 a1'); g.is_free()
False
```

preorder_traversal()
Yields words as if traversing the vertices of a tree in pre-order.

simple_words_above()
An iterator that yields the simple words which can be obtained by contracting this basis $n \geq 0$ times. (So the 'above' condition is not strict.)

```
>>> g = Generators((3, 1), ["x a1 a1 a3", "x a1 a1 a1", "x a1 a2", "x a1 a1 a2
↪", "x a1 a3"])
>>> for word in g.simple_words_above():
...     print(format(word))
x1 a1 a1 a1
x1 a1 a1 a2
x1 a1 a1 a3
x1 a1 a2
x1 a1 a3
x1 a1 a1
x1 a1
```

test_generates_algebra()
Tests to see if this set generates all of $V_{n,r}$. The generating set is sorted, and then contracted as much as possible. Then we check to see which elements of the *standard basis* $\boldsymbol{x} = \{x_1, \ldots, x_r\}$ are not included in the contraction.

The test fails if there is at least one such element; in this case, the list of all such elements is returned. Otherwise the test passes, and an empty list is returned.

```
>>> #A basis for V_2,1
>>> Y = Generators((2, 1), ["x a1", "x a2 a1", "x a2 a2 a1 a1", "x a2 a2 a1 a2
↪", "x a2 a2 a2"])
>>> Y.test_generates_algebra()
[]
>>> #The same words do not generate V_2,3
>>> Y = Generators((2, 3), ["x a1", "x a2 a1", "x a2 a2 a1 a1", "x a2 a2 a1 a2
↪", "x a2 a2 a2"])
>>> missing = Y.test_generates_algebra(); missing
[Word('x2', (2, 3)), Word('x3', (2, 3))]
>>> for x in missing: print(x)
x2
x3
```

Return type a list of *Words*.

generates_algebra()
Returns True if this set generates all of $V_{n,r}$. Otherwise returns False.

```
>>> from random import randint
>>> arity = randint(2, 5); alphabet_size = randint(2, 10)
>>> Y = Generators.standard_basis((arity, alphabet_size))
>>> Y.generates_algebra()
True
>>> random_basis().generates_algebra()
True
```

test_above(*word*, *return_index=False*)
Searches for a pair (gen, tail) where gen belongs to the current basis and gen + tail = word. If no such pair exists, returns None.

If *return_index* is False, returns the pair *(gen, tail)*. Otherwise, returns the pair *(i, tail)* where *i* is the index of *gen* in the current basis.

```
>>> basis = Generators.standard_basis((2, 1)).expand(0).expand(0).expand(0)
>>> basis
Generators((2, 1), ['x1 a1 a1 a1', 'x1 a1 a1 a2', 'x1 a1 a2', 'x1 a2'])
>>> gen, tail = basis.test_above(Word('x1 a2 a2 a1', (2, 1)))
>>> print(gen, '|', format(tail))
x1 a2 | a2 a1
>>> i, tail = basis.test_above(Word('x1 a2 a2 a1', (2, 1)), return_index=True)
>>> print(basis[i])
x1 a2
>>> basis.test_above(Word('x1', (2,1))) is None
True
>>> gen, tail = basis.test_above(basis[0])
>>> print(gen)
x1 a1 a1 a1
>>> gen is basis[0] and len(tail) == 0
True
```

is_above(*word*)
Returns True if any generator *is_above()* the given *word*.

```
>>> g = Generators((2, 2), ['x1 a1', 'x1 a2', 'x2'])
>>> g.is_above(Word('x1 a1 a1 a2', (2, 2)))
True
>>> g.is_above(Word('x1', (2, 2)))
False
>>> g.is_above(Word('x2', (2, 2)))
True
>>> g.is_above(Word('x1 a1 x1 a2 L', (2, 2)))
False
```

is_basis()
Returns True if this is a *free generating set* which *generates all of* $V_{n,r}$. Otherwise returns False.

```
>>> g = Generators((2, 2), ["x1 a1", "x1 a2"])
>>> g.is_free()
True
>>> g.generates_algebra()
False
>>> g.is_basis()
```

(continues on next page)

(continued from previous page)

```
False
>>> g.append('x2')
>>> g.is_basis()
True
>>> random_basis().is_basis()
True
```

descendants_above (*floor*)

Suppose we have a basis *floor* below the current basis *ceiling*. There are a finite number of elements below *ceiling* which are not below *floor*. This method enumerates them. In symbols, we are enumerating the set $X\langle A\rangle \setminus F\langle A\rangle$, where X is the current basis and F is the *floor*.

```
>>> phi = load_example('example_5_15')
>>> X = phi.quasinormal_basis
>>> Y = X.minimal_expansion_for(phi)
>>> Z = phi.image_of_set(Y)
>>> terminal = X.descendants_above(Y)
>>> initial  = X.descendants_above(Z)
>>> print(initial, terminal, sep='\n')
[x1 a1, x1 a1 a1]
[x1 a2 a2, x1 a2 a2 a1]
```

classmethod standard_basis (*signature*)

Creates the standard basis $\boldsymbol{x} = \{x_1, \ldots, x_r\}$ for $V_{n,r}$, where n is the arity and r is the *alphabet_size* of the *signature*.

```
>>> Generators.standard_basis((2, 4))
Generators((2, 4), ['x1', 'x2', 'x3', 'x4'])
```

classmethod from_dfs (*string*)

Creates bases correponding to binary trees via a string of 1s and 0s correponding to branches and carets respectively. See *from_dfs()*. The *string* can also be specified as a `int`, in which case it is replaced by its base 10 string representation.

```
>>> print(Generators.from_dfs("100"))
[x1 a1, x1 a2]
>>> print(Generators.from_dfs(1100100)) #ints are fine too
[x1 a1 a1, x1 a1 a2, x1 a2 a1, x1 a2 a2]
```

Raises **ValueError** – if the *string* doesn't correctly describe a rooted binary tree.

```
>>> print(Generators.from_dfs(""))
Traceback (most recent call last):
...
ValueError: Should have one more zero than one (got 0 and 0 respectively)
>>> print(Generators.from_dfs("10"))
Traceback (most recent call last):
...
ValueError: Should have one more zero than one (got 1 and 1 respectively)
>>> print(Generators.from_dfs(10001))
Traceback (most recent call last):
...
ValueError: Error at character 3: complete description of tree with␣
↪unprocessed digits remaining. String was 10001
```

minimal_expansion_for(**automorphisms*)

Suppose we are given a finite sequence of *automorphisms* of $V_{n,r}$ and that the current generaeting set is a basis X for $V_{n,r}$. This methods returns an expansion Y of X such that each automorphism maps Y into $X\langle A\rangle$.

```
>>> std_basis = Generators.standard_basis((2, 1))
>>> reduced_domain = Generators((2, 1), ["x a1 a1", "x a1 a2 a1", "x a1 a2 a2
↪", "x a2 a1", "x a2 a2"])
>>> std_basis.minimal_expansion_for(load_example('cyclic_order_six')) ==␣
↪reduced_domain
True
```

```
>>> phi = load_example('example_5_3')
>>> basis = phi.quasinormal_basis
>>> print(basis.minimal_expansion_for(phi))
[x1 a1 a1 a1 a1, x1 a1 a1 a1 a2, x1 a1 a1 a2, x1 a1 a2 a1, x1 a1 a2 a2, x1 a2␣
↪a1, x1 a2 a2 a1, x1 a2 a2 a2]
```

Raises

- **ValueError** – if no automorphisms are passed to this method.
- **ValueError** – if the automorphisms or basis do not all belong to the same group $G_{n,r}$.
- **ValueError** – if the given basis does not generate $V_{n,r}$.

See also:

Lemma 4.3 of the paper proves that this expansion exists, is of minimal size, and is unique with this that property.

embed_in(*signature*, *shift=0*)

Creates a copy of the current generating set in the algebra determined by *signature*.

Raises **ValueError** – if the current signature's algebra is not a subset of the given signature's algebra.

```
>>> x = Generators.standard_basis((2, 1)).expand(0); x
Generators((2, 1), ['x1 a1', 'x1 a2'])
>>> y = x.embed_in((3, 2)); y
Generators((3, 2), ['x1 a1', 'x1 a2'])
>>> z = x.embed_in((3, 2), shift=1); z
Generators((3, 2), ['x2 a1', 'x2 a2'])
>>> print(y.signature, y[0].signature)
(3, 2) (3, 2)
>>> x.is_basis()
True
>>> y.is_basis()
False
```

expand(*index*)

Replaces the word w at index *index* in the current generating set with its children, $w\alpha 1, \ldots, w\alpha_n$, where n is the arity of the generating set. As with ordinary Python lists, negative values of *index* are supported.

```
>>> g = Generators.standard_basis((3, 1)); g
Generators((3, 1), ['x1'])
>>> g.expand(0)
Generators((3, 1), ['x1 a1', 'x1 a2', 'x1 a3'])
```

(continues on next page)

(continued from previous page)

```
>>> g #has been modified
Generators((3, 1), ['x1 a1', 'x1 a2', 'x1 a3'])
>>> g.expand(-2) #expand at the second entry from the right, i.e. at 'x1 a2'
Generators((3, 1), ['x1 a1', 'x1 a2 a1', 'x1 a2 a2', 'x1 a2 a3', 'x1 a3'])
```

Raises `IndexError` – if there is no generator at index *index*.

Returns the current generating set, after modification.

expand_away_lambdas()

Experimental method to recursively (and inefficiently) expand a generating set until every word has lambda-length 0.

```
>>> X = Generators((2,1), ['x1 a1 a1 a1 a2', 'x1 a2 a1 a2 x1 a2 a1 a1 a2 x1␣
↪a2 a2 a2 L x1 a2 a2 a1 L L'])
>>> X.expand_away_lambdas()
>>> print(X)
[x1 a1 a1 a1 a2, x1 a2 a1 a2, x1 a2 a1 a1 a2, x1 a2 a2 a2, x1 a2 a2 a1]
```

classmethod sort_mapping_pair(*domain*, *range*)

Makes copies of the given lists of words *domain* and *range*. The copy of *domain* is sorted according to the *order* defined on words. The same re-ordering is applied to the *range*, so that the mapping from *domain* to *range* is preserved.

Return type A pair of class:*Generator* instances.

expand_to_size(*size*)

Expands the current generating set until it has the given *size*. The expansions begin from the end of the generating set and work leftwards, wrapping around if we reach the start. (This is to try and avoid creating long words where possible.)

```
>>> basis = Generators.standard_basis((3, 1)); print(basis)
[x1]
>>> basis.expand_to_size(11); print(basis)
[x1 a1 a1, x1 a1 a2, x1 a1 a3, x1 a2 a1, x1 a2 a2, x1 a2 a3, x1 a3 a1, x1 a3␣
↪a2, x1 a3 a3 a1, x1 a3 a3 a2, x1 a3 a3 a3]
>>> basis = Generators.standard_basis((2, 1)); print(basis)
[x1]
>>> basis.expand_to_size(2); print(basis)
[x1 a1, x1 a2]
>>> basis = Generators.standard_basis((2, 3)).expand(2).expand(0);␣
↪print(basis)
[x1 a1, x1 a2, x2, x3 a1, x3 a2]
>>> basis.expand_to_size(12); print(basis)
[x1 a1 a1, x1 a1 a2, x1 a2 a1, x1 a2 a2, x2 a1, x2 a2, x3 a1 a1, x3 a1 a2, x3␣
↪a2 a1 a1, x3 a2 a1 a2, x3 a2 a2 a1, x3 a2 a2 a2]
```

Expanding a basis to its current size does nothing.

```
>>> b1 = random_basis()
>>> b2 = b1.copy()
>>> b2.expand_to_size(len(b1))
>>> b1 == b2
True
```

If expansion to the target size is not possible, a `ValueError` is raised.

```
>>> basis = Generators.standard_basis((3,2))
>>> len(basis)
2
>>> basis.expand_to_size(3)
Traceback (most recent call last):
...
ValueError: Cannot expand from length 2 to length 3 in steps of size 2.
>>> basis.expand_to_size(4)
>>> len(basis)
4
>>> basis.expand_to_size(1)
Traceback (most recent call last):
...
ValueError: Cannot expand from length 4 to length 2 in steps of size 2.
```

cycled(*positions=1*)

Produces a copy of the current generating set, cyclicly shifted by a number of *positions* to the right.

```
>>> phi = load_example('example_5_15')
>>> print(phi.domain)
[x1 a1, x1 a2 a1, x1 a2 a2 a1 a1, x1 a2 a2 a1 a2, x1 a2 a2 a2]
>>> print(phi.domain.cycled(2))
[x1 a2 a2 a1 a1, x1 a2 a2 a1 a2, x1 a2 a2 a2, x1 a1, x1 a2 a1]
```

```
>>> X = random_basis()
>>> X == X.cycled(0) == X.cycled(len(X))
True
```

thompson.generators.**rewrite_set**(*set*, *basis*, *new_basis*)

Maps *set* into the algebra generated by *new_basis* according to the bijection between *basis* and *new_basis*. See also *rewrite_word()*.

thompson.generators.**rewrite_word**(*word*, *basis*, *new_basis*)

Suppose that we have a *word* which is below a given *basis*. Suppose we have a bijection between *basis* and some image *new_basis*. Then we can rewrite *word* as as descendant of *new_basis* in a manner which is compatible with this bijection.

Raises

- **ValueError** – if *basis* is not above *word*.
- **ValueError** – if *basis* and *new_basis* have different sizes.

3.4.2 Next steps

Suppose we have a map $f\colon V_{n,r} \to V_{n',r'}$. If this is just a set map then we would have to specify the image of every word in $V_{n,r}$. However, if f preserves structure then it is sufficient to specify the image of a generating set—or even better, a basis—for $V_{n,r}$. Such maps f are called *homomorphisms*. The next module describes homomorphisms in terms of the image of a given basis.

3.5 Homomorphisms

The algebra of *words* has its own structure, and just like groups, rings etc. there exist homomorphisms which preserve this structure. In our specific case, a homomorphism $\phi : V_{n,r} \to V_{n,s}$ is a function satisfying

$$w\alpha_i\phi = w\phi\alpha_i \qquad \text{and} \qquad w_1 \dots w_n\lambda\phi = aw_1\phi \dots w_n\phi\lambda.$$

In other words, a homomorphism is a function which 'commutes' with the algebra operations α_i and λ.

3.5.1 The Homomorphism Class

class thompson.homomorphism.**Homomorphism**(*domain*, *range*, *reduce=True*)

Let $f : D \to R$ be some map embedding a basis D for $V_{n,r}$ into another algebra $V_{n,s}$ of the same *Signature*. This map uniquely extends to a homomorphism of algebras $\psi : V_{n,r} \to V_{n,s}$.

Variables

- **domain** – a *basis* of preimages
- **range** – a *basis* of images.

The next few attributes are used internally when constructing free factors. We need to have a way to map back to the parent automorphisms.

Variables

- **domain_relabeller** – the homomorphism which maps relabelled words back into the original algebra that this automorphism came from.
- **range_relabeller** – the same.

__init__(*domain*, *range*, *reduce=True*)

Creates a homomorphism with the specified *domain* and *range*. Sanity checks are performed so that the arguments do genuinely define a basis.

By default, any redudancy in the mapping is reduced. For example, the rules `x1 a1 a1 -> x1 a2 a1` and `x1 a1 a2 -> x1 a2 a2` reduce to the single rule `x1 a1 -> x1 a2`. The *reduce* argument can be used to disable this. This option is intended for internal use only.[1]

Raises

- **TypeError** – if the bases are of different sizes.
- **TypeError** – if the algebras described by *domain* and *range* have different arities.
- **ValueError** – if *domain* is *not a basis*.

classmethod from_file(*filename*)

Reads in a file specifying a homomorphism and returns the homomorphism being described. Here is an example of the format:

```
5
(2,1)           ->      (2,1)
x1 a1 a1 a1     ->      x1 a1 a1
x1 a1 a1 a2     ->      x1 a1 a2 a1
x1 a1 a2        ->      x1 a1 a2 a2
x1 a2 a1        ->      x1 a2 a2
x1 a2 a2        ->      x1 a2 a1
This is an example intended to illustrate the format for reading and writing␣
↪automorphisms to disk.
```

(continues on next page)

[1] Sometimes we have to expand *free factors* so that the orbit sizes match up.

(continued from previous page)

There are three components

- number k of generators in domain and range
- signatures of domain and range
- list of k rules domain -> range

Any lines after this are ignored, and can be treated as comments. Comments are read in and added to the __doc__ attribute of the homomorphism that gets created.

Todo: Should use a different attribute (.comment?) rather than __doc__.

classmethod from_string(*string*)
An alternative *from_file()* for working with examples that you might want to copy and paste somewhere. *string* should be a Python string which describes an automorphism in the same format as in *from_file()*.

static next_non_comment_line(*file*)

save_to_file(*filename=None*, *comment=None*)
Takes a homomorphism and saves it to the file with the given *filename*. The homomorphism is stored in a format which is compatible with *from_file()*. Optionally, a *comment* may be appended to the end of the homomorphism file.

```
>>> before = random_automorphism()
>>> before.save_to_file('test_saving_homomorphism.aut')
>>> after = Automorphism.from_file('test_saving_homomorphism.aut')
>>> before == after, before is after
(True, False)
```

__eq__(*other*)
We can test for equality of homomorphisms by using Python's equality operator ==. The Python integer `1` can be used as a shorthand for the identity of the current homomorphism's algebra.

```
>>> phi = random_automorphism()
>>> phi == phi
True
>>> phi * ~phi == 1
True
>>> 1 == Homomorphism.identity(phi.signature)
True
```

__mul__(*other*)
If the current automorphism is ψ and the *other* is ϕ, multiplication forms the composition $\psi\phi$, which maps $x \mapsto x\psi \mapsto (x\psi)\phi$.

Raises **TypeError** – if the homomorphisms cannot be composed in the given order; i.e. if `self.range.signature != other.domain.signature`.

Return type an *MixedAut* if possible; otherwise a *Homomorphism*.

```
>>> phi = load_example('alphabet_size_two')
>>> print(phi * phi)
MixedAut: V(3, 2) -> V(3, 2) specified by 8 generators (after expansion and␣
↪reduction).
```

(continues on next page)

(continued from previous page)

```
x1 a1    -> x1 a1
x1 a2    -> x1 a2 a3 a3
x1 a3 a1 -> x1 a3
x1 a3 a2 -> x1 a2 a2
x1 a3 a3 -> x1 a2 a1
x2 a1    -> x2
x2 a2    -> x1 a2 a3 a2
x2 a3    -> x1 a2 a3 a1
```

classmethod identity(*signature*)

Creates a new homo/automorphism which is the identity map on the algebra with the given *signature*.

```
>>> print(Homomorphism.identity((3, 2)))
PeriodicAut: V(3, 2) -> V(3, 2) specified by 2 generators (after expansion␣
↪and reduction).
x1 -> x1
x2 -> x2
>>> sig = random_signature()
>>> Homomorphism.identity(sig) == 1
True
```

is_identity()

Returns True if this automorphism is the identity map on the algebra with the given *signature*. Otherwise returns False.

```
>>> aut = Homomorphism.identity(random_signature())
>>> aut.is_identity()
True
>>> load_example('example_5_15').is_identity()
False
```

image(*key*, *sig_in=None*, *sig_out=None*, *cache=None*)

Computes the image of a *key* under the given homomorphism. The result is cached for further usage later.

The *key* must be given as one of:

- a string to be passed to *from_string()*;
- a sequence of integers (see the *word* module); or
- a *Word* instance.

Note: All other arguments are intended for internal use only.[2]

The input need not be in standard form. This method

1. Converts *key* to standard form if necessary.
2. Checks if the image of the standard form of *key* has been cached, and returns the image if so.
3. Computes the image of *key* under the homomorphism, then caches and returns the result.

```
>>> #An element of the domain---just a lookup
>>> phi = load_example('example_5_15')
>>> print(phi.image('x1 a1'))
```

(continues on next page)

[2] Exposing more arguments means I can write this function in a more general manner. Doing so makes it easy to compute *inverse images* under an *MixedAut*.

(continued from previous page)

```
x1 a1 a1 a1
>>> #A word below a the domain words
>>> print(phi.image('x1 a1 a2 a2'))
x1 a1 a1 a1 a2 a2
>>> #Above domain words---have to expand.
>>> print(phi.image('x1'))
x1 a1 a1 a1 x1 a1 a1 a2 x1 a2 a2 x1 a1 a2 L x1 a2 a1 L L L
>>> #Let's try some words not in standard form
>>> print(phi.image('x1 a1 a1 x1 a1 a2 L'))
x1 a1 a1 a1
>>> print(phi.image('x1 a1 a1 x1 a1 a2 L a2 a1'))
x1 a1 a1 a1 a2 a1
>>> print(phi.image('x1 a1 a1 x1 a2 a2 L'))
x1 a1 a1 a1 a1 x1 a2 a2 x1 a1 a2 L x1 a2 a1 L L
```

Return type a *Word* instance (which are always in standard form).

image_of_set(*set*, *sig_in=None*, *sig_out=None*, *cache=None*)

Computes the image of a list of *Generators* under the current homomorphism. The order of words in the list is preserved.

Return type another list of *Generators*.

```
>>> basis = Generators.standard_basis((2,1))
>>> basis.expand_to_size(8); print(basis)
[x1 a1 a1 a1, x1 a1 a1 a2, x1 a1 a2 a1, x1 a1 a2 a2, x1 a2 a1 a1, x1 a2 a1 a2,
↪ x1 a2 a2 a1, x1 a2 a2 a2]
>>> print(load_example('example_5_3').image_of_set(basis))
[x1 a1 a1 a1 x1 a1 a1 a2 a1 L, x1 a1 a1 a2 a2, x1 a1 a2 a2, x1 a1 a2 a1, x1␣
↪a2 a1 a1 a1, x1 a2 a1 a1 a2, x1 a2 a1 a2, x1 a2 a2]
```

image_of_point(*x*)

Interpret the current homomorphism as a piecewise linear map and evalutate the image of x under this map. Note that inverse images aren't supported, as homomorphisms need not be invertible.

Parameters **x** (*Fraction*) – the current point, as an `int` or a `Fraction`.

```
>>> from fractions import Fraction
>>> x = standard_generator(0)
>>> x.image_of_point(0)
Fraction(0, 1)
>>> x(0)    #can also just use __call__ syntax
Fraction(0, 1)
>>> x.image_of_point(Fraction(1, 3))
Fraction(1, 6)
```

__call__(**args*, ***kwargs*)

Homomorphisms are callable. Treating them as a function just calls the *image()* method.

```
>>> from fractions import Fraction
>>> x0 = standard_generator()
>>> x0(Fraction(1, 2))
Fraction(1, 4)
```

__str__()

Printing an automorphism gives its arity, alphabet_size, and lists the images of its domain elements.

```
>>> print(load_example('cyclic_order_six'))
PeriodicAut: V(2, 1) -> V(2, 1) specified by 5 generators (after expansion␣
↪and reduction).
x1 a1 a1    -> x1 a1 a1
x1 a1 a2 a1 -> x1 a1 a2 a2 a2
x1 a1 a2 a2 -> x1 a2
x1 a2 a1    -> x1 a1 a2 a2 a1
x1 a2 a2    -> x1 a1 a2 a1
```

__repr__()
A tweaked version of *__str__()*. This produces a string representation which can be passed to *from_string()* to reobtain the Homomorphism we started with.

```
>>> f = random_automorphism()
>>> g = Automorphism.from_string( repr(f) )
>>> f == g
True
```

classmethod pl_segment(*d1*, *d2*, *r1*, *r2*)

pl_segments()

format_pl_segments(*LaTeX=False*, *sfrac=False*)
Returns a description of the current homomorphism as a piecewise-linear map on the interval.

```
>>> x = standard_generator()
>>> print(x.format_pl_segments())
0   + 1/2 (t - 0   ) from 0   to 1/2
1/4 + 1   (t - 1/2 ) from 1/2 to 3/4
1/2 + 2   (t - 3/4 ) from 3/4 to 1
```

Parameters

- **LaTeX** (*bool*) – if True, the description is formatted as a LaTeX `cases` environment.
- **sfrac** (*bool*) – if True, and if *LaTeX* is True, format fractions using xfrac's `\sfrac` command.

```
>>> print(x.format_pl_segments(LaTeX=True))
%\usepackage{array}
\begin{equation*}
        \setlength\arraycolsep{1.3pt}
        \left\{ \begin{array}{ccrl}c<{\quad}ccc}
                 0   &+& 1/2 &(t - 0   ) &\text{if}& 0   &\leq t < & 1/2  \\
                 1/4 &+& 1   &(t - 1/2 ) &\text{if}& 1/2 &\leq t < & 3/4  \\
                 1/2 &+& 2   &(t - 3/4 ) &\text{if}& 3/4 &\leq t < & 1
        \end{array} \right.
\end{equation*}
```

classmethod format_pl_segments_directly(*segments*, *LaTeX=False*, *sfrac=False*)

tikz_path()
Return a string which can be passed to a `\tikz\draw` command to graph the current homomorphism.

add_relabellers(*domain_relabeller*, *range_relabeller*)

Raises

- **LookupError** – if a relabeller is provided and the relabeller's signature does not match that of *domain* or *range*
- **TypeError** – if a relabeller is provided and the relabeller's signature does not match that of *domain* or *range* (as appropriate)

relabel()

If this automorphism was derived from a parent automorphism, this converts back to the parent algebra after doing computations in the derived algebra.

In the following example *test_conjugate_to()* takes a pure periodic automorphism and extracts factors. A conjugator ρ is produced by *the overridden version of this method*. Finally ρ is relabelled back to the parent algebra.

Raises **AttributeError** – if the factor has not been assigned any relabellers.

```
>>> psi, phi = load_example_pair('example_5_12')
>>> rho = psi.test_conjugate_to(phi)
>>> print(rho)
PeriodicAut: V(2, 1) -> V(2, 1) specified by 6 generators (after expansion␣
↪and reduction).
x1 a1 a1 a1 a1 -> x1 a1 a2
x1 a1 a1 a1 a2 -> x1 a2 a2
x1 a1 a1 a2    -> x1 a1 a1 a1
x1 a1 a2       -> x1 a2 a1 a1
x1 a2 a1       -> x1 a1 a1 a2
x1 a2 a2       -> x1 a2 a1 a2
>>> rho * phi == psi * rho
True
```

gradients()

Interprets the current homomorphism as a piecewise linear map, and returns the list of gradients of each linear piece. The i th element of this list is the gradient of the affine map sending `self.domain[i]` to `self.range[i]`.

Return type `list` of `Fraction` s.

```
>>> standard_generator(0).gradients()
[Fraction(1, 2), Fraction(1, 1), Fraction(2, 1)]
>>> load_example("alphabet_size_two").gradients()
[Fraction(1, 1), Fraction(1, 3), Fraction(3, 1), Fraction(1, 1), Fraction(1,␣
↪3), Fraction(1, 3)]
```

gradient_at(x)

Interprets the current homomorphism as a piecewise linear map, and returns the right-derivative of the current homomorphism at x.

Return type `Fraction`

```
>>> f = standard_generator(0)
>>> f.gradient_at(0)
Fraction(1, 2)
>>> f.gradient_at(-1)
Traceback (most recent call last):
...
AssertionError
>>> f.gradient_at(2/3)
Fraction(1, 1)
>>> f.gradient_at(Word("x a2", (2, 1)))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: Automorphism doesn't map x1 a2 affinely
>>> f.gradient_at(Word("x a2 a2 a2 a2", (2, 1)))
Fraction(2, 1)
```

static gradient(*domain*, *range*)

Computes the gradient of the affine map sending the interval represented by *domain* to the interval represented by *range*.

3.5.2 Next steps

It is not possible to repeatedly apply a homomorphism ψ to a word unless ψ is actually an automorphism. The next class extends *Homomorphism* to represent an *Automorphism*.

3.6 Automorphisms

As usual, a bijective homomorphism is called an *isomorphism*, and an isomorphism from an algebra to itself is called an *automorphism*. A collection of automorphisms of an object O forms a group $\mathrm{Aut}(O)$ under composition. The group of automorphisms $\mathrm{Aut}(V_{n,r})$ is known as $G_{n,r}$.

We consider three different classes of automorphism; this module implements functionality common to each class.

3.6.1 The Automorphisms class

class thompson.automorphism.**Automorphism**(*domain*, *range*, *reduce=True*)

Represents an automorphism as a bijection between *bases*.

Generic attributes:

Variables

- **signature** – The *Signature* shared by the generating sets domain and range.
- **quasinormal_basis** – See *compute_quasinormal_basis()*.
- **pond_banks** – A list of tuples (ℓ, k, r) such that (ℓ, r) are banks of a pond with $\ell\phi^k = r$. (See also 4.15 of the paper.)

```
>>> olga_f = load_example('olga_f')
>>> olga_g = load_example('olga_g')
>>> olga_f.signature
Signature(arity=2, alphabet_size=1)
>>> print(len(olga_f.pond_banks)) #No ponds
0
>>> print(len(olga_g.pond_banks)) #One pond
1
>>> print(*olga_g.pond_banks[0], sep=',  ')
x1 a2 a1 a2 a1 a1,  2,  x1 a1 a2 a2
```

Periodic attributes:

Variables

- **cycle_type** – the set $\{d \in \mathbb{N} : \exists$ an orbit of length $d.\}$. This is implemented as a `frozenset` which is a readonly version of Python's `set` type.
- **order** – The *`lcm()`* of the automorphism's cycle type. This is the group-theoretic order of the *`periodic factor`* of ϕ. If the cycle type is empty, the order is ∞. **NB**: *`mixed automorphisms`* will have a **finite** order, despite being infinite-order group elements.
- **periodic_orbits** – a mapping $d \mapsto L_d$, where L_d is the list of size d orbits in the quasinormal basis.
- **multiplicity** – a mapping $d \mapsto m_\phi(d, X_\phi)$, which is the number of periodic orbits of size d in the *`quasinormal basis`* for ϕ. See Definition 5.8 in the paper.

```
>>> olga_f.cycle_type
frozenset()
>>> olga_f.order
inf
>>> f = load_example('cyclic_order_six')
>>> f.cycle_type
frozenset({1, 2, 3})
>>> f.order
6
>>> def display_orbits(orbits_by_size):
...     for key in sorted(orbits_by_size):
...             print('Orbits of length', key)
...             for list in orbits_by_size[key]:
...                     print('...', *list, sep=' -> ', end=' -> ...\n')
>>> display_orbits(load_example('example_5_9').periodic_orbits)
Orbits of length 2
... -> x1 a2 a1 a1 -> x1 a2 a1 a2 -> ...
... -> x1 a2 a2 a1 -> x1 a2 a2 a2 -> ...
Orbits of length 3
... -> x1 a1 a1 a1 -> x1 a1 a1 a2 -> x1 a1 a2 -> ...
```

Infinite attributes:

Variables **characteristics** – the set of characteristics (m, Γ) of this automorphism. Like the `cycle_type` above, this is a `frozenset`.

```
>>> for char in sorted(olga_f.characteristics):
...             #TODO the order here isn't sorted intuitively
...     print(char)
Characteristic(-1, a2)
Characteristic(-1, a1)
Characteristic(2, a2 a1)
Characteristic(2, a1 a2)
```

__init__(*domain*, *range*, *reduce=True*)

Creates an automorphism mapping the given *domain* basis to the *range* basis in the given order. That is, domain$_i \mapsto$ range$_i$.

The *`quasi-normal basis`* X and the various attributes are all calculated at creation time.

Raises

- **TypeError** – if the bases have different arities or alphabet sizes.
- **TypeError** – if either basis *`isn't actually a basis`*.

See also:

`The superclass' method`.

classmethod from_dfs(*domain*, *range*, *labels=None*, *reduce=True*)

Creates elements of $V = G_{2,1}$ using the notation of *[Kogan]*.

The domain and range trees are described as a string of ones and zeros. A `1` denotes a vertex which has children; a `0` denotes a vertex which has none (i.e. a leaf). The 'dfs' stands for *depth-first search*, the order in which we traverse the vertices of the tree. We initialise the current vertex as the root. Upon reading a `1` we add two child vertices to the current vertex, then redeclare the current vertex to be the current vertex's left child. Upon reading a `0` we set the current vertex to be the next vertex of the tree in depth-first order. If the current vertex is the root we are done.

Parameters

- **domain** (*str*) – A description of a binary tree as a stream of ones and zeroes.
- **range** (*str*) – The same.
- **labels** (*str*) – A string of natural numbers $1, \ldots, m$ in some order. If omitted, taken to be the string `"1 2 "..."len(domain)"`. If a single string `n`, taken to be the cyclic permuation mapping $1 \mapsto n$.
- **reduce** (*bool*) – Passed to *the superclass' initialiser method*. If `True`, carets are reduced in the domain and range where possible.

```
>>> f = Automorphism.from_dfs("100", "100", "2 1")
>>> f.order
2
>>> print(f)
PeriodicAut: V(2, 1) -> V(2, 1) specified by 2 generators (after expansion␣
↪and reduction).
x1 a1 -> x1 a2
x1 a2 -> x1 a1
>>> Automorphism.from_dfs("100", "100", "2") == f
True

>>> g = Automorphism.from_dfs("1010100", "1011000")
>>> print(g)
MixedAut: V(2, 1) -> V(2, 1) specified by 4 generators (after expansion and␣
↪reduction).
x1 a1       -> x1 a1
x1 a2 a1    -> x1 a2 a1 a1
x1 a2 a2 a1 -> x1 a2 a1 a2
x1 a2 a2 a2 -> x1 a2 a2
>>> g == standard_generator(1)
True

>>> h = load_example("example_6_2")
>>> h2 = Automorphism.from_dfs("1110000", "1100100", "3 1 2 4")
>>> h == h2
True
```

classmethod rotation(*numerator*, *denominator=None*)

Constructs an automorphism which represents a rotation when thought of as a homeomorphism of the circle. The *displacement* of this rotation be a dyadic rational. If both arguments are given, the displacement is `numerator/denominator`; if only one argument is given, the displacement is the result of passing *numerator* to the constructor of `Fraction`.

Note: This only works in $T_{2,1}$, but could be modified in future to work for different arities.

Raises `ValueError` – if the displacement is not dyadic.

Returns *PeriodicAut*

```
>>> print(Automorphism.rotation(1, 2))
PeriodicAut: V(2, 1) -> V(2, 1) specified by 2 generators (after expansion␣
↪and reduction).
x1 a1 -> x1 a2
x1 a2 -> x1 a1
>>> print(Automorphism.rotation(3, 8))
PeriodicAut: V(2, 1) -> V(2, 1) specified by 8 generators (after expansion␣
↪and reduction).
x1 a1 a1 a1 -> x1 a1 a2 a2
x1 a1 a1 a2 -> x1 a2 a1 a1
x1 a1 a2 a1 -> x1 a2 a1 a2
x1 a1 a2 a2 -> x1 a2 a2 a1
x1 a2 a1 a1 -> x1 a2 a2 a2
x1 a2 a1 a2 -> x1 a1 a1 a1
x1 a2 a2 a1 -> x1 a1 a1 a2
x1 a2 a2 a2 -> x1 a1 a2 a1
>>> Automorphism.rotation(0) == Automorphism.rotation(100) == Automorphism.
↪identity((2, 1))
True

>>> denominator = 2 ** randint(1, 10)
>>> numerator = randrange(denominator)
>>> rot = Automorphism.rotation(numerator, denominator)
>>> rot.rotation_number() == Fraction(numerator, denominator)
True
>>> rot.cycles_order()
True
>>> rot.preserves_order() == (numerator == 0)
True
```

image (*key*, *inverse=False*)

If *inverse* is True, the inverse of the current automorphism is used to map *key* instead. Otherwise this method delegates to *Homomorphism.image*.

As a shorthand, we can use function call syntax in place of this method. For instance:

```
>>> phi = load_example('example_5_15')
>>> print(phi('x1 a2 a2 a1 a1'))
x1 a2 a2
```

Examples of finding inverse images:

```
>>> phi = load_example('example_5_15')
>>> print(phi.image('x1 a2 a2', inverse=True))
x1 a2 a2 a1 a1
>>> print(phi.image('x1 a1 a1 a2 a2 a1', inverse=True))
x1 a2 a1 a2 a1
>>> print(phi.image('x a2', inverse=True))
x1 a2 a2 a2 x1 a2 a2 a1 a1 L
>>> print(phi.image('x a2 a2 x a1 a2 L', inverse=True))
x1 a2 a2 a1
```

image_of_set (*set*, *inverse=False*)

If *inverse* is True, the inverse of the current automorphism is used to map *set* instead. Otherwise this method delegates to *Homomorphism.image_of_set*.

Examples of finding inverse images:

```
>>> basis = Generators.standard_basis((2,1))
>>> basis.expand_to_size(8);
>>> print(load_example('example_5_3').image_of_set(basis, inverse=True))
[x1 a1 a1 a1 a1, x1 a1 a1 a1 a2 x1 a1 a1 a2 L, x1 a1 a2 a2, x1 a1 a2 a1, x1␣
↪a2 a1, x1 a2 a2 a1, x1 a2 a2 a2 a1, x1 a2 a2 a2 a2]
```

repeated_image(*key*, *power*)

If ψ is the current automorphism, returns keyψ^{power}.

Return type a *Word* instance.

```
>>> phi = load_example('example_5_15')
>>> print(phi.repeated_image('x1 a1', 10))
x1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1
>>> print(phi.repeated_image('x1 a1 a1 a1 a1 a1 a1 a1', -3))
x1 a1
>>> phi = load_example('arity_four')
>>> print(phi.repeated_image('x1 a4 a4 a2', 4))
x1 a3 a3 a2
>>> print(phi.repeated_image('x1 a3 a3 a2', -4))
x1 a4 a4 a2
```

Todo: This could be made more efficient for large values of *power* by using knowledge of the component containing *key*.

__pow__(*power*)

Python uses the double star operator to denote exponentiation.

```
>>> psi = random_automorphism()
>>> psi ** 0 == 1 #identity
True
>>> ~psi ** 2 == ~psi * ~psi == psi ** -2
True
```

__invert__()

We overload python's unary negation operator ~ as shorthand for inversion. (In Python, ~ is normally used for bitwise negation.) We can also call a method explicitly: `phi.inverse()` is exactly the same as `~phi`.

```
>>> phi = random_automorphism()
>>> phi * ~phi == ~phi * phi
True
>>> (phi * ~phi).is_identity()
True
>>> (~phi).quasinormal_basis == phi.quasinormal_basis
True
```

__xor__(*other*)

We overload the bitwise exclusive or operator ^ as a shorthand for conjugation. We act on the right, so that $s^c = c^{-1}sc$.

```
>>> sig = random_signature()
>>> f = random_automorphism(signature=sig)
>>> g = random_automorphism(signature=sig)
```

(continues on next page)

(continued from previous page)

```
>>> h = f^g
>>> f.is_conjugate_to(h)
True
>>> f ^ (f.test_conjugate_to(h)) == h
True
```

compute_quasinormal_basis()

We say that ϕ is *in semi-normal form* with respect to the basis X if no element of X lies in an incomplete X-component of a ϕ orbit. See the *orbits* module for more details.

There is a minimal such basis, X_ϕ say, and we say that ϕ is *in quasi-normal form* with respect to X_ϕ. This method determines and returns the basis X_ϕ where ϕ denotes the current automorphism. The result is cached so that further calls to this method perform no additional computation.

Note: This method is called automatically at creation time and does **not** need to be called by the user.

```
>>> for name in ['example_4_5', 'alphabet_size_two', 'example_5_12_phi',
↪'example_6_2', 'example_6_8_phi']:
...     print(load_example(name).quasinormal_basis)
[x1 a1 a1, x1 a1 a2, x1 a2 a1, x1 a2 a2]
[x1 a1, x1 a2, x1 a3, x2]
[x1 a1, x1 a2]
[x1 a1 a1, x1 a1 a2, x1 a2]
[x1 a1, x1 a2]
```

Return type a *Generators* instance.

See also:

Quasi-normal forms are introduced in Section 4.2 of the paper. In particular, this method implements Lemma 4.28. Higman first described the idea of quasi-normal forms in Section 9 of *[Hig74]*.

seminormal_form_start_point()

Returns the minimal expansion X of $\boldsymbol{x}$ such that every element of X belongs to either *self.domain* or *self.range*. Put differently, this is the minimal expansion of $\boldsymbol{x}$ which does not contain any elements which are above $Y \cup Z$. This basis that this method produces is the smallest possible which *might* be semi-normal.

```
>>> for name in ['example_4_5', 'example_4_11', 'example_4_12', 'example_5_15
↪', 'cyclic_order_six']:
...     print(load_example(name).seminormal_form_start_point())
[x1 a1 a1, x1 a1 a2, x1 a2 a1, x1 a2 a2]
[x1 a1, x1 a2]
[x1 a1, x1 a2]
[x1 a1, x1 a2 a1, x1 a2 a2]
[x1 a1 a1, x1 a1 a2 a1, x1 a1 a2 a2, x1 a2]
```

See also:

Remark 4.10 of the paper.

orbit_type(*y*, *basis=None*)

Returns the orbit type of *y* with respect to the given *basis*. If *basis* is omitted, the *quasi-normal basis* is used by default. Also returns a dictionary of computed images, the list (4.6) from the paper.

Returns A triple `(ctype, images, type_b_data)`.

```
>>> phi = load_example('example_4_5')
>>> print_component_types(phi, phi.domain)
x1 a1 a1 a1: Left semi-infinite component with characteristic (-1, a1)
x1 a1 a1 a2: Bi-infinite component
x1 a1 a2: Right semi-infinite component with characteristic (1, a2)
x1 a2 a1: Periodic component of order 2
x1 a2 a2: Periodic component of order 2
with respect to the basis [x1 a1 a1 a1, x1 a1 a1 a2, x1 a1 a2, x1 a2 a1, x1␣
↪a2 a2]
>>> print_component_types(phi, basis=phi.domain, words=['x', 'x a1', 'x a2'])
x1: Incomplete component
x1 a1: Incomplete component
x1 a2: Incomplete component
with respect to the basis [x1 a1 a1 a1, x1 a1 a1 a2, x1 a1 a2, x1 a2 a1, x1␣
↪a2 a2]
```

Components can be calculated with respect to any *basis*, not just the *quasi-normal basis*.

```
>>> print(olga_f.quasinormal_basis)
[x1 a1 a1, x1 a1 a2, x1 a2 a1, x1 a2 a2 a1, x1 a2 a2 a2]
>>> basis = olga_f.quasinormal_basis.copy().expand(-1); print(basis)
[x1 a1 a1, x1 a1 a2, x1 a2 a1, x1 a2 a2 a1, x1 a2 a2 a2 a1, x1 a2 a2 a2 a2]
>>> from pprint import pprint
>>> print(olga_f.orbit_type('x a2 a2 a2')[0])
Bi-infinite component
>>> print(olga_f.orbit_type('x a2 a2 a2', basis)[0])
Incomplete component
```

See also:

Lemmas 4.18, 4.28 of the paper.

Todo: This should be renamed to component_type.

---

test_semi_infinite(*y*, *basis*, *backward=False*)

Computes the orbit type of *y* under the current automorphism ψ with respect to *basis* in the given direction. Let $y\psi^m$ be the most recently computed image. The process stops when either:

1. $y\psi^m$ is not below the *basis*, for some $m \geq 0$.
   - infinite: `False`
   - start: `0`
   - end: `m-1`
   - images: $y, y\psi, \ldots, y\psi^{m-1}$.
2. For some $0 \leq l < m$, $y\psi^l$ is above (or equal to) $y\psi^m$.
   - infinite: `True`
   - start: `l`
   - end: `m`
   - images: $y, y\psi, \ldots, y\psi^m$.

Note: The word $y\psi^m$ is not strictly in the core part of the orbit of Lemma 4.24. We return this as part of *images* so that we can compute the characteristic multiplier in *orbit_type()*.

Returns the tuple *(infinite, start, end, images)*.

See also:

Lemma 4.28 of the paper.

semi_infinite_end_points(*exclude_characteristics=False*)

Returns the list of terminal *Words* in left semi-infinite components and the list of initial words in right semi-infinite components. This is all computed with respect the current automorphism's *quasinormal basis*. These are the sets $X\langle A\rangle \setminus Y\langle A\rangle$ and $X\langle A\rangle \setminus Z\langle A\rangle$.

Parameters **exclude_characteristics** – if True, only the endpoints of non-characteristic semi-infinite orbits will be returned.

```
>>> for id in ['4_5', '4_11', '4_12', '5_15', '6_2']:
...     aut = load_example('example_' + id)
...     print(*aut.semi_infinite_end_points())
[x1 a1 a1] [x1 a1 a2]
[x1 a1] [x1 a2]
[] []
[x1 a2 a2, x1 a2 a2 a1] [x1 a1, x1 a1 a1]
[x1 a1 a1] [x1 a2]
>>> phi = load_example('nathan_pond_example')
>>> print(*phi.semi_infinite_end_points())
[x1 a1 a1, x1 a1 a1 a1, x1 a1 a1 a2] [x1 a1 a2, x1 a1 a2 a1, x1 a1 a2 a2]
>>> print(*phi.semi_infinite_end_points(exclude_characteristics=True))
[x1 a1 a1 a2] [x1 a1 a2 a2]
```

Return type A pair of *Generators*.

See also:

The discussion before Lemma 4.6.

dump_QNB()

A convenience method for printing the quasinormal basis X. The X-components of the elements $x \in X$ are displayed.

```
>>> load_example('example_5_3').dump_QNB()
x1 a1 a1 a1 Left semi-infinite component with characteristic (-1, a1)
x1 a1 a1 a2 Right semi-infinite component with characteristic (1, a2)
x1 a1 a2 a1 Periodic component of order 2
x1 a1 a2 a2 Periodic component of order 2
x1 a2 a1 Right semi-infinite component with characteristic (1, a1)
x1 a2 a2 Left semi-infinite component with characteristic (-1, a2)
```

dump_periodic(*cantor=False*)

A convenience method for printing out periodic orbits in the quasinormal basis.

```
>>> f = load_example("cyclic_order_six")
>>> f.dump_periodic()
Period 1
x1 a1 a1
```

(continues on next page)

(continued from previous page)

```
Period 2
x1 a1 a2 a2 a1 -> x1 a2 a1
Period 3
x1 a1 a2 a1 -> x1 a1 a2 a2 a2 -> x1 a2 a2
>>> f.dump_periodic(cantor=True)
Period 1
00
Period 2
0110 -> 10
Period 3
010 -> 0111 -> 11
```

print_characteristics()

For convenience, a method that prints out all of the characteristics of type B components wrt the quasinormal_basis.

```
>>> for id in ['4_1', '5_15', '6_2', '6_8_phi']:
...     name = 'example_' + id
...     print(name)
...     load_example(name).print_characteristics()
example_4_1
(-1, a1)
(1, a2)
example_5_15
(-1, a1 a1)
(1, a1 a1)
example_6_2
(-2, a1)
(1, a2)
example_6_8_phi
(-1, a1 a1 a1)
(1, a2 a2 a2)
>>> (load_example('example_6_2')**2).print_characteristics()
(-1, a1)
(1, a2 a2)
>>> #Lemma 5.16
>>> psi, phi = random_conjugate_pair()
>>> psi.characteristics == phi.characteristics
True
```

See also:

Defintion 5.14.

share_orbit(u, v)

Determines if u and v are in the same orbit of the current automorphism ψ. Specifically, does there exist an integer m such that $u\psi^m = v$?

```
>>> phi = load_example('alphabet_size_two')
>>> u  = Word('x1 a2 a3 a1 a2', (3, 2))
>>> v1 = Word('x1 a1 a2 a2 a3 a1', (3, 2))
>>> v2 = Word('x2 a3 a2', (3, 2))
>>> print(phi.share_orbit(u, v1))
{}
>>> print(phi.share_orbit(u, v2))
{-2}
>>> print(phi.share_orbit(u, u))
```

(continues on next page)

(continued from previous page)

```
{0}
```

Returns The (possibly empty) *SolutionSet* of all integers m for which $u\psi^m = v$. Note that if $u = v$ this method returns $\mathbb{Z}$.

See also:

The implementation is due to lemma 4.34 of the paper.

is_conjugate_to(*other*)
A shortcut for `self.test_conjugate_to(other) is not None`.

preserves_order()
Returns True if this automorphism is an element of $F_{n,r}$, Higman's analogue of Thompson's group F. Otherwise returns False.

```
>>> random_automorphism(group='F').preserves_order()
True
>>> phi = random_automorphism(group='T')
>>> #phi preserves order iff it is in F.
>>> (sorted(phi.range) == phi.range) == phi.preserves_order()
True
>>> load_example('nathan_pond_example').preserves_order()
False
```

cycles_order()
Returns True if this automorphism is an element of $T_{n,r}$, Higman's analogue of Thompson's group T. Otherwise returns False.

```
>>> random_automorphism(group='F').cycles_order()
True
>>> random_automorphism(group='T').cycles_order()
True
>>> load_example('nathan_pond_example').cycles_order() # in V \ T
False
```

rotation_number()
The *rotation number* is a map $\mathrm{Homeo}_+(\mathbb{S}^1) \to \mathbb{R}/\mathbb{Z}$ which is constant on conjugacy classes.

Return type `Fraction` if the current automorphism is a *circle homeomorphism*; otherwise *None*.

```
>>> f = random_automorphism()
>>> rot = f.rotation_number()
>>> (rot is None) == (not f.cycles_order())
True
>>> sig = random_signature()
>>> Automorphism.identity(sig).rotation_number() == 0
True
>>> g = random_automorphism(group='T', signature=sig)
>>> k = random_automorphism(group='T', signature=sig)
>>> (g^k).rotation_number() == g.rotation_number()
True

>>> f_examples = "example_4_1 example_4_11 example_6_8_psi example_6_9_psi␣
↪non_dyadic_fixed_point".split()
>>> all( load_example(name).rotation_number() == 0 for name in f_examples )
```

(continues on next page)

(continued from previous page)

```
True
>>> t_examples = "example_4_1 example_5_12_phi example_6_8_phi non_dyadic_
↪fixed_point rotation_number_one_third scott_free_alpha thirds_fixed".split()
>>> for name in t_examples:
...     print(load_example(name).rotation_number())
0
1/2
0
0
1/3
0
0
```

commutes(*other*)

A shorthand for the expression `self * other == other * self`.

```
>>> x = random_automorphism()
>>> x.commutes(x)
True
>>> x.commutes(~x)
True
>>> x.commutes(1)
True
>>> x.commutes(x*x)
True
```

test_revealing(*domain='wrt QNB'*)

Determines if the given automorphism ϕ is revealed by the tree pair $(D, \phi(D))$, where D is the given *domain*.

The *domain* may be implicitly specified by the string `'minimal'` or `'wrt QNB'`. In the first case, *domain* is taken to be the minimal *domain* required to specify the automorphism. In the second case, *domain* is taken to be the minimal expansion of the quasinormal basis.

Returns *None* if the pair is reveling for ϕ. Otherwise, returns (as a *Word*) the root of a component of either $D \setminus \phi(D)$ or $\phi(D) \setminus D$ which does not contain an attractor/repeller.

```
>>> load_example('olga_f').test_revealing() is None
True
>>> print(load_example('semi_inf_c').test_revealing())
x1 a2 a1
>>> print(load_example('non_revealing').test_revealing())
x1 a1 a1
>>> f = load_example('cyclic_order_six')
>>> print(f.test_revealing('minimal'))
x1 a2
>>> f.test_revealing('wrt QNB') is None
True
```

Caution: This is an experimental feature based on *[SD10]*.

Todo: This method is badly named; something like `test_revealed_by(basis)` would be better.

is_revealing(*domain='wrt QNB'*)

Calls *test_revealing()*, but only returns `True` or `False`.

Returns True if the pair is revealing, otherwise False.

```
>>> load_example('olga_f').is_revealing()
True
>>> load_example('semi_inf_c').is_revealing()
False
>>> load_example('non_revealing').is_revealing()
False
>>> f = load_example('cyclic_order_six')
>>> f.is_revealing('minimal')
False
>>> f.is_revealing('wrt QNB')
True
>>> load_example('v_revealing_test').is_revealing()
False
```

Caution: This is an experimental feature based on *[SD10]*.

Todo: This method is badly named; something like `is_revealed_by(basis)` would be better.

periodic_points(*include_intervals=True*)

Identifies the intervals and points which are (pointwise) peridically mapped under the current automorphism. Pointwise periodic intervals are represented as :class‘~thompson.word.Word‘s; isolated periodic points are represented as `Fraction` s.

Parameters **include_intervals** (*bool*) – If False, intervals are not included in the output. If the current automorphism is in T, the result is the boundary of the set of fixed points.

Returns an `OrderedDict` mapping periodic objects to their period size.

Caution: This is an experimental feature and provides different information to *fixed_points()*.

fixed_points()

Returns a list of `Fraction` and :class‘~thompson.word.Word‘s which are the fixed points and intervals of this function.

```
>>> print( *Automorphism.identity((2, 1)).fixed_points() , sep=" ")
x1
>>> standard_generator().fixed_points()
[Fraction(0, 1), Fraction(1, 1)]
>>> f = load_example("non_dyadic_fixed_point")
>>> f.fixed_points()
[Fraction(0, 1), Fraction(1, 3), Word('x1 a2 a2', (2, 1))]
>>> x = Automorphism.from_dfs("1101000", "1100100", "2 3 4 1")
>>> x.fixed_points()
[Fraction(1, 2)]
>>> y = Automorphism.from_dfs("1101000", "1011000", "2 3 4 1")
>>> y.fixed_points()
[Fraction(1, 3), Fraction(2, 3)]
```

```
>>> f = random_automorphism()
>>> fixed_intervals = (x for x in f.fixed_points() if isinstance(x, Word))
>>> all( f(interval) == interval for interval in fixed_intervals )
True
```

Caution: This is an experimental feature.

fixed_point_boundary(*on_circle=False*)

Replaces any intervals in the output of *fixed_points()* with their endpoints.

```
>>> f = load_example("non_dyadic_fixed_point")
>>> f.fixed_point_boundary()
[Fraction(0, 1), Fraction(1, 3), Fraction(3, 4), Fraction(1, 1)]
```

Parameters **on_circle** (*bool*) – if True, treat 0 and 1 as the same point.

```
>>> f.fixed_point_boundary(on_circle=True)
[Fraction(0, 1), Fraction(1, 3), Fraction(3, 4)]
```

area_to_identity(*scaled=False*)

Let $\phi \in F_{n,1}$, viewed as a bijection of the interval. What is the (unsigned) area between ϕ and the identity?

Parameters **scaled** (*bool*) – If true, the area is normalised by ignoring subintervals where the function is equal to the identity.

Warning: This is an experimental feature based on a suggestion by Henry Bradford.

```
>>> for n in range(4):
...     print(standard_generator(n).area_to_identity())
...
5/32
5/128
5/512
5/2048
>>> x0 = standard_generator(0)
>>> x1 = standard_generator(1)
>>> g = ~x0 * x1 * x0
>>> print(g.area_to_identity())
3/32
>>> f = load_example('non_dyadic_fixed_point')
>>> print(f.area_to_identity())
43/768

>>> for n in range(4):
...             print(standard_generator(n).area_to_identity(scaled=True))
5/32
5/32
5/32
5/32
```

centralise_period(*period*, *trees*, *rearranger*)

Constructs a new element ψ commuting with the given element ϕ. We construct the centralising element

by altering the ϕ-orbit structure below the orbits of the given *period*.

There are two parameters: a collection of labelled *trees* and a *rearranger* element; in the notation of *[BBG11]* these are elements of K_{m_i} and G_{n,r_i} respectively.

Todo: doctests

Caution: This is an experimental feature based on *[BBG11]*.

gradients_around(*x*)

Let $x \in \mathbb{Q} \cap [0,1]$. What's the gradient to the left of x and right of x? If x is a breakpoint it could be different; if not, it'll be the same number on both sides.

Return type 2-tuple of fractions.Fraction.

```
>>> f = standard_generator(0)
>>> f.gradients_around(5/6)
(Fraction(2, 1), Fraction(2, 1))
>>> f.gradients_around(1/2)
(Fraction(1, 2), Fraction(1, 1))
>>> f.gradients_around(0)
(Fraction(2, 1), Fraction(1, 2))

>>> g = load_example("alphabet_size_two")
>>> g.gradients_around(1/2)
(Fraction(3, 1), Fraction(1, 1))
>>> g.gradients_around(1/3)
(Fraction(1, 3), Fraction(1, 3))
>>> #One third is a breakpoint, but floating point can't accurately represent␣
↪one third. Need to use a Fraction here
>>> g.gradients_around(Fraction(1, 3))
(Fraction(1, 3), Fraction(3, 1))
>>> g.gradients_around(0)
(Fraction(1, 3), Fraction(1, 1))
```

Membership placeholders

As an alternative to using the *preserves_order()* and *cycles_order()* methods, we provide two objects `F`, `T` which know how to respond to the `in` operator.

```
>>> f = random_automorphism(group='F')
>>> f in F and f in T
True
>>> t = load_example('scott_free_alpha')
>>> t not in F and t in T
True
```

3.6.2 Next steps

Because an Automorphism can be *repeatedly applied*, we may consider the orbit $\{w\psi^n \mid n \in \mathbb{Z}\}$ of any word w. (Note that this is the orbit of w under the cyclic subgroup $\langle\psi\rangle$, rather than all of $G_{n,r}$.)

The next module gives us tools to classify and work with these orbits. (In truth, the `Automorphism` class uses these tools, so this documentation is really in the wrong order.)

3.7 Orbits

Let y be a word, and let X be an expansion of the standard basis $\boldsymbol{x} = \{x_1, \ldots, x_n\}$; finally let ϕ be an `MixedAut`. We call the set $\mathrm{orb}_\phi(y) = \{y\phi^i\}_{i\in\mathbb{Z}}$ the ϕ-orbit of y. Let us refer to the intersection $\mathrm{orb}_\phi(y) \cap X\langle A\rangle$ as the X *-component* of (the ϕ-orbit of) y. Higman demonstrated how we could classify these components in *[Hig74]* (section 9).

This module is responsible for two orbit-related data structures. Firstly, `ComponentType` is set up to describe all possible types of orbits according to Higman's scheme. Secondly, the `SolutionSet` describes solutions to the equation $u\phi^i = v$.

3.7.1 Types of orbits and components

These components come in five different types:

1. **Complete infinite components.** The component is the entire orbit $\{y\phi^i\}_{i\in\mathbb{Z}}$ and each element of the orbit is different.
2. **Complete finite components.** The component is the entire orbit, which eventually it repeats itself. Thus the component only contains a finite number of distinct words.
3. **Right semi-infinite components.** The forward orbit $\{y\phi^n\}_{n\in\mathbb{N}}$ belongs to the component and does not repeat itself. However, the backward orbit $\{y\phi^{-n}\}_{n\in\mathbb{N}}$ eventually leaves $X\langle A\rangle$; thus the component only contains a finite amount of the backward orbit.
4. **Left semi-infinite components.** The backward orbit is contained in the component, and does not repeat itself; the forward orbit eventually leaves $X\langle A\rangle$.
5. **Incomplete finite components.** A finite part of the orbit

$$y\phi^{-n}, \ldots, y\phi^{-1}, y, y\phi, \ldots, y\phi^m$$

for which both $y\phi^{-n-1}$ and $y\phi^{m+1}$ are not in $X\langle A\rangle$.

There are six different ways we can form orbits using these components:

- **Complete infinite (or *doubly infinite*) orbits.** The entire orbit is a complete infinite component (type 1).
- **Complete finite (or *periodic*) orbits.** The entire orbit is a component finite component (type 2).
- **Incomplete finite (or *bad*) orbits.** The orbit does not contain any complete or semi-infinite components. Thus the only components which may appear are of type 5. Note that these orbits need not have any components at all!
- **Incomplete infinite orbits.** The orbit contains at least one semi-infinite component (types 3, 4). The orbit may also contain incomplete finite components (type 5). We give names to the different cases:
  - **Right semi-infinite orbits.** One component of type 3, none of type 4.
  - **Left semi-infinite orbits.** One component of type 4, none of type 3.
  - ***Pond* orbits.** One component of type 3 and one of type 4. These components must be separated by a finite list of words in $V_{n,r} \setminus X\langle A\rangle$; we think of this list as being a *pond*.

If that wasn't confusing enough, we have a another way to classify those orbits which are not incomplete finite.

1. Type A components are those with characteristic (m, ϵ), where ϵ is the empty word. These are precisely the periodic orbits.

2. Type B components are those with characteristic (m, Γ), where $\Gamma \neq \epsilon$ is nontrivial.
3. Type C components are those which are not of type A or B—that is, they are the components which do not have a characteristic. If x belongs to a type C orbit, then $x\phi^\ell = z\Delta$ for some z of type B. That is, type C orbits are those 'below' type B orbits.

Note: Type B components must be semi-infinite, **but not all semi-infinite components are of type B**. Complete infinite components are always of type C, but **some semi-infinite components are of type C too**.

---

class thompson.orbits.**ComponentType**

This class is essentially a glorified `enumeration`: its instances store a number from 1 to 5 to represent the type of a component.

Variables **characteristic** – Either the characteristic (m, Γ) of this component, or `None` if this component has no characteristic. Periodic components have characteristic (m, ε), where ε is the empty word.

See also:

Section 4.1 of the paper.

classmethod periodic(*period*)

Describes a complete finite (i.e. periodic, type 2) component. The argument is the period of the the component.

classmethod semi_infinite(*characteristic*, *backward=False*)

Describes a semi-infinite component (types 3, 4). The first argument is the characteristic of the component; use `None` if this component doesn't have a characteristic. The second argument should specify where this component is left or right semi-infinite.

classmethod complete_infinite()

Describes a component which is infinite in both directions (type 1).

classmethod incomplete()

Describes a component which is incomplete finite (type 5).

is_type_A()

Returns true if this component belongs to an orbit of type A (periodic).

is_type_B()

Returns true if this component belongs to an orbit of type B (has a characteristic)

is_type_C()

Returns true if this component belongs to an orbit of type C (does not have a characteristic)

is_incomplete()

Returns True if this component is incomplete, otherwise False.

is_semi_infinite()

Returns True is this component is semi_infinite (with or without characteristic); otherwise returns False.

3.7.2 The SolutionSet structure

Solutions to the equation $u\psi^m = v$ come in specific instances. If there are no solutions, the solution set is $\emptyset$. Otherwise u and v share an orbit. If this orbit is periodic, then the solution set is of the form $m + p\mathbb{Z}$, where p is the period of the orbit. Otherwise the solution set is a single integer m.

Internally we represent this as a pair *(base, increment)* of integers. The first element *base* is a solution m if it exists; otherwise `None`. The second element *increment* is the increment p between solutions (which occurs for a periodic orbit only).

class thompson.orbits.**SolutionSet**

Solution sets to the orbit sharing test $u\phi^k = v$. These are either

- empty (no such k) exists;
- a singleton; or
- a linear sequence a + b*n.

Create solution sets as follows:

```
>>> print(SolutionSet(5, 2))
{..., 3, 5, 7, 9, 11, 13, ...}
>>> print(SolutionSet.singleton(4))
{4}
>>> print(SolutionSet.empty_set())
{}
>>> print(SolutionSet.the_integers())
{..., -1, 0, 1, 2, 3, 4, ...}
```

is_sequence()

Returns True if this set contains more than one distinct element; otherwise returns False.

```
>>> SolutionSet.empty_set().is_sequence()
False
>>> SolutionSet.singleton(2).is_sequence()
False
>>> SolutionSet(base=4, increment=3).is_sequence()
True
>>> SolutionSet.the_integers().is_sequence()
True
```

is_singleton()

Returns True if this contains precisely one element, otherwise False.

```
>>> SolutionSet.empty_set().is_singleton()
False
>>> SolutionSet.singleton(2).is_singleton()
True
>>> SolutionSet(base=4, increment=3).is_singleton()
False
>>> SolutionSet.the_integers().is_singleton()
False
```

is_empty()

Returns True if this set is empty, otherwise False.

```
>>> SolutionSet.empty_set().is_empty()
True
>>> SolutionSet.singleton(2).is_empty()
False
>>> SolutionSet(base=4, increment=3).is_empty()
False
>>> SolutionSet.the_integers().is_empty()
False
```

is_the_integers()
Returns True if this set contains every integer; otherwise returns False.

```
>>> SolutionSet.empty_set().is_the_integers()
False
>>> SolutionSet.singleton(2).is_the_integers()
False
>>> SolutionSet(base=4, increment=3).is_the_integers()
False
>>> SolutionSet.the_integers().is_the_integers()
True
```

__contains__(*other*)
Returns true if this set contains an *other* number.

```
>>> 1024 in SolutionSet.empty_set()
False
>>> 1024 in SolutionSet.singleton(128)
False
>>> 1024 in SolutionSet(0, 256)
True
>>> 1024 in SolutionSet.the_integers()
True
```

__and__(*other*)
The `&` operator (usually used for bitwise and) stands for intersection of sets.

```
>>> phi = SolutionSet.empty_set()
>>> Z = SolutionSet.the_integers()
>>> singleton = SolutionSet.singleton
>>> print(phi & phi)
{}
>>> print(phi & singleton(1))
{}
>>> print(phi & SolutionSet(2, 3))
{}
>>> print(singleton(1) & singleton(1))
{1}
>>> print(singleton(1) & singleton(2))
{}
>>> print(singleton(8) & SolutionSet(4, 2))
{8}
>>> print(SolutionSet(1, 3) & SolutionSet(2, 3))
{}
>>> print(SolutionSet(1, 3) & SolutionSet(1, 2))
{..., -5, 1, 7, 13, 19, 25, ...}
>>> print(SolutionSet(1, 18) & SolutionSet(5, 24))
{}
>>> print(SolutionSet(1, 18) & SolutionSet(13, 24))
{..., -35, 37, 109, 181, 253, 325, ...}
>>> print(SolutionSet(1, 3) & Z)
{..., -2, 1, 4, 7, 10, 13, ...}
>>> print(Z & Z)
{..., -1, 0, 1, 2, 3, 4, ...}
```

3.7.3 Helper functions

thompson.orbits.**print_component_types**(*aut*, *basis=None*, *words=None*)

Prints the classification of the components under *aut* of each word in *words* with respect to *basis*. If *basis* is omitted, it is taken to be the smallest possible expansion which could potentially be a semi-normal basis; see *seminormal_form_start_point()*. If *words* is omited, it is taken to be the same as *basis*.

```
>>> print_component_types(load_example('arity_three_order_inf'))
x1 a1: Left semi-infinite component with characteristic (-1, a1)
x1 a2: Bi-infinite component
x1 a3 a1: Bi-infinite component
x1 a3 a2: Bi-infinite component
x1 a3 a3: Right semi-infinite component with characteristic (1, a1)
with respect to the basis [x1 a1, x1 a2, x1 a3 a1, x1 a3 a2, x1 a3 a3]
```

```
>>> print_component_types(load_example('arity_four'))
x1 a1 a1: Periodic component of order 4
x1 a1 a2: Periodic component of order 4
x1 a1 a3: Periodic component of order 4
x1 a1 a4: Periodic component of order 4
x1 a2: Bi-infinite component
x1 a3: Right semi-infinite component with characteristic (1, a3)
x1 a4: Left semi-infinite component with characteristic (-1, a4)
with respect to the basis [x1 a1 a1, x1 a1 a2, x1 a1 a3, x1 a1 a4, x1 a2, x1 a3,␣
↪x1 a4]
```

See also:

The *orbit_type()* method.

class thompson.orbits.**Characteristic**(*power*, *multiplier*)

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *power*, *multiplier*)
Create new instance of Characteristic(power, multiplier)

__repr__()
Return a nicely formatted representation string

multiplier
Alias for field number 1

power
Alias for field number 0

3.7.4 Next steps

With tools to describe orbits in hand, we can dive straight into the *Automorphism* class. In particular, we need to *know how orbits work* to

- determine *quasinormal bases*
- perform the *orbit sharing test*
- perform the *conjugacy test*

3.8 MixedAuts

A general automorphism $\phi \in G_{n,r}$ may exhibit periodic behaviour, infinite behaviour, or a mix of both. We say that an automorphism is

- *purely periodic* if it has finite order;
- *purely infinite* if it has no periodic *orbits*; and
- *mixed* otherwise.

We can represent a mixed automorphism ψ as a free product $\psi = \psi_P * \psi_I$ of a purely periodic and purely infinite automorphism. The automorphisms ψ_P and ψ_I are called *free_factors()*. We form this decomposition because the conjugacy problem is easier to solve when the automorphisms are both pure infinite or both pure periodic.

This class is responsible for:

- Generating the free factors from a mixed automorphism;
- Combining a periodic and infinite factor into a mixed automorphism; and
- Delegating the conjugacy and power conjugacy tests to these factors.

3.8.1 The MixedAut class

class thompson.mixed.**MixedAut**(*domain*, *range*, *reduce=True*)

free_factors()

In principle, an automorphism can be decomposed into free factors using any partition of the *quasi-normal basis* $X = X_1 \sqcup X_2$. The decomposition $X = X_P \sqcup X_I$ is particularly interesting since these sets generate ψ-invariant subalgebras, which means that a conjugator can be reassembled from conjugators of the factors.

See also:

This jargon comes from Theorem 5.1.

This is a convenience method which produces the free factors ψ_P and ψ_I from ψ.

free_factor(*generators*)

This method restricts the current automorphism to the subalgebra generated by the given set W of *generators*. This is then transformed into an automorphism of an isomorphic algebra with minimal alphabet size $1 \leq s \leq n-1$.

$$\begin{aligned} & G_{n,r} \\ \to & G_{n,|W|} \\ \to & G_{n,s} \\ & \phi \\ \mapsto & \phi|_{W\langle A\rangle\langle\lambda\rangle} \\ \mapsto & \phi' \end{aligned}$$

Returns The transformed automorphism $\phi \in G_{n,s}$. Its type is *PeriodicAut* or *InfiniteAut* as appropriate.

Raises **ValueError** – if an empty list of *generators* is provided.

```
>>> phi = load_example('alphabet_size_two')
>>> qnb = phi.quasinormal_basis
>>> p, i = phi._partition_basis(qnb)
>>> print(phi.free_factor(p))
PeriodicAut: V(3, 1) -> V(3, 1) specified by 1 generators (after expansion␣
↪and reduction).
This automorphism was derived from a parent automorphism.
'x' and 'y' represent root words of the parent and current derived algebra,␣
↪respectively.
x1 a1 ~>    y1 => y1    ~> x1 a1
>>> print(phi.free_factor(i))
InfiniteAut: V(3, 1) -> V(3, 1) specified by 5 generators (after expansion␣
↪and reduction).
This automorphism was derived from a parent automorphism.
'x' and 'y' represent root words of the parent and current derived algebra,␣
↪respectively.
x1 a2 ~>    y1 a1    => y1 a1 a3    ~> x1 a2 a3
x1 a3 ~>    y1 a2    => y1 a3       ~> x2
x2 a1 ~>    y1 a3 a1 => y1 a2       ~> x1 a3
x2 a2 ~>    y1 a3 a2 => y1 a1 a2    ~> x1 a2 a2
x2 a3 ~>    y1 a3 a3 => y1 a1 a1    ~> x1 a2 a1
```

test_conjugate_to(*other*)

Determines if the current automorphism ψ is conjugate to the *other* automorphism ϕ.

Returns if it exists, a conjugating element ρ such that $\rho^{-1}\psi\rho = \phi$. If no such ρ exists, returns `None`.

Raises **ValueError** – if the automorphisms have different arities or alphabet sizes.

```
>>> psi, phi = random_conjugate_pair()
>>> rho = psi.test_conjugate_to(phi)
>>> rho is not None
True
>>> psi * rho == rho * phi
True
>>> psi, phi = load_example_pair('first_pond_example')
>>> rho = psi.test_conjugate_to(phi)
>>> rho is not None
True
>>> psi * rho == rho * phi
True
```

See also:

This is an implementation of Algorithm 5.6 in the paper. It depends on Algorithms 5.13 and 5.27; these are the *periodic* and *infinite* tests for conjugacy.

find_power_conjugators(*other*, *cheat=False*)

Determines if some power of the current automorphism ψ is conjugate to some power of the *other* automorphism ϕ. This method exhaustively searches for all minimal solutions (a, b, ρ) such that $\rho^{-1}\psi^a\rho = \phi^b$. This method returns a generator which yields such minimal solutions.

Warning: If the *power_conjugacy_bounds()* are reasonable large (say > 30), this method could potentially take a long time!

Parameters **cheat** – (internal) for speeding up testing.

Raises `ValueError` – if the automorphisms have different arities or alphabet sizes.

See also:

This is an implementation of Algorithm 6.13 in the paper. It depends on Algorithms 5.6 (the *conjugacy test*) and 6.11 (the *infinite power conjugate test*.)

test_power_conjugate_to(*other*, *cheat=False*)

Determines if some power of the current automorphism ψ is conjugate to some power of the *other* automorphism ϕ. This method searches for a **single** minimal solution (a, b, ρ) such that $\rho^{-1}\psi^a\rho = \phi^b$. If no such solution exists, returns None.

> **Warning:** If the *power_conjugacy_bounds()* are reasonable large (say > 30), this method could potentially take a long time!

Parameters `cheat` – (internal) for speeding up testing.

Raises `ValueError` – if the automorphisms have different arities or alphabet sizes.

```
>>> psi, phi = load_example_pair('mixed_pconj')
>>> a, b, rho = psi.test_power_conjugate_to(phi)
>>> a, b
(6, 3)
>>> psi**a * rho == rho * phi ** b
True
```

See also:

This is an implementation of Algorithm 6.13 in the paper. It depends on Algorithms 5.6 (the *conjugacy test*) and 6.11 (the *infinite power conjugate test*.)

3.8.2 Next steps

To complete the details of the *conjugacy test*, we have to be able to test if two pure periodic automorphisms are conjugate and if two pure infinite automorphisms are conjugate. Any given automorphism can be broken down into a pure periodic and pure infinite part. These parts are called *free factors*.

3.9 Free Factors and conjugacy

To *test for conjugacy* we need to extract periodic and infinite components of an automorphism and test those components for conjugacy. These next few classes keep track of

- how the components embed into the original automorphism, and
- any extra information that we need to test the components for conjugacy.

3.9.1 Periodic Factors

class thompson.periodic.**PeriodicAut**(*domain*, *range*, *reduce=True*)

A purely periodic automorphism, which may have been extracted from a mixed automorphism.

```
>>> example_5_9 = load_example('example_5_9')
>>> print(example_5_9)
PeriodicAut: V(2, 1) -> V(2, 1) specified by 7 generators (after expansion and␣
↪reduction).
x1 a1 a1 a1 -> x1 a1 a1 a2
x1 a1 a1 a2 -> x1 a1 a2
x1 a1 a2    -> x1 a1 a1 a1
x1 a2 a1 a1 -> x1 a2 a1 a2
x1 a2 a1 a2 -> x1 a2 a1 a1
x1 a2 a2 a1 -> x1 a2 a2 a2
x1 a2 a2 a2 -> x1 a2 a2 a1
>>> sorted(example_5_9.cycle_type)
[2, 3]
>>> #Two orbits of size 2, one orbit of size 3
>>> from pprint import pprint
>>> pprint(example_5_9.multiplicity)
{2: 2, 3: 1}
>>> example_5_9.order
6
```

```
>>> load_example('cyclic_order_six').order
6
>>> phi = random_periodic_automorphism()
>>> 1 <= phi.order < float('inf')
True
>>> len(phi.cycle_type) != 0
True
>>> len(phi.characteristics)
0
```

test_conjugate_to(*other*)

We can determine if two purely periodic automorphisms are conjugate by examining their orbits.

```
>>> psi_p, phi_p = load_example('example_5_12_psi'), load_example('example_5_
↪12_phi')
>>> rho_p = psi_p.test_conjugate_to(phi_p)
>>> print(rho_p)
PeriodicAut: V(2, 1) -> V(2, 1) specified by 6 generators (after expansion␣
↪and reduction).
x1 a1 a1 a1 a1 -> x1 a1 a2
x1 a1 a1 a1 a2 -> x1 a2 a2
x1 a1 a1 a2    -> x1 a1 a1 a1
x1 a1 a2       -> x1 a2 a1 a1
x1 a2 a1       -> x1 a1 a1 a2
x1 a2 a2       -> x1 a2 a1 a2
>>> rho_p * phi_p == psi_p * rho_p
True
>>> psi_p, phi_p = random_conjugate_periodic_pair()
>>> rho_p = psi_p.test_conjugate_to(phi_p)
>>> rho_p * phi_p == psi_p * rho_p
True
```

See also:

This implements algorithm 5.13 of the paper—see section 5.3.

find_power_conjugators(*other*, *identity_permitted=False*, *cheat=False*)

test_power_conjugate_to(*other*, *cheat=False*)

Tests two periodic factors to see if they are power conjugate. Yields minimal solutions (a, b, ρ). such that $\rho^{-1}\psi^a\rho = \phi^b$.

See also:

Section 6.1 of the paper.

power_conjugacy_bounds(*other*, *identity_permitted*)

We simply try all powers of both automorphisms. There are only finitely many, because everything is periodic.

Returns `self.power, other.power` if the identity is permitted as a solution; otherwise `self.power - 1, other.power - 1`.

Todo: Maybe this should be 0 to order if the identity is permitted and 1 to order otherwise?

See also:

Section 6.1 of the paper.

3.9.2 Infinite Factors

class thompson.infinite.**InfiniteAut**(*domain*, *range*, *reduce=True*)

A purely infinite automorphism which may have been extracted from a mixed automorphism.

```
>>> print(load_example('example_5_3').free_factors()[1])
InfiniteAut: V(2, 1) -> V(2, 1) specified by 6 generators (after expansion and␣
↪reduction).
This automorphism was derived from a parent automorphism.
'x' and 'y' represent root words of the parent and current derived algebra,␣
↪respectively.
x1 a1 a1 a1 a1 ~>    y1 a1 a1 a1 => y1 a1 a1       ~> x1 a1 a1 a1
x1 a1 a1 a1 a2 ~>    y1 a1 a1 a2 => y1 a1 a2 a1    ~> x1 a1 a1 a2 a1
x1 a1 a1 a2    ~>    y1 a1 a2    => y1 a1 a2 a2    ~> x1 a1 a1 a2 a2
x1 a2 a1       ~>    y1 a2 a1    => y1 a2 a1 a1    ~> x1 a2 a1 a1
x1 a2 a2 a1    ~>    y1 a2 a2 a1 => y1 a2 a1 a2    ~> x1 a2 a1 a2
x1 a2 a2 a2    ~>    y1 a2 a2 a2 => y1 a2 a2       ~> x1 a2 a2
>>> phi = random_infinite_automorphism()
>>> phi.order
inf
>>> len(phi.cycle_type)
0
>>> len(phi.characteristics) != 0
True
```

test_conjugate_to(*other*)

We can determine if two purely infinite automorphisms are conjugate by breaking down the quasi-normal basis into *equivalence classes*.

```
>>> for psi_i, phi_i in (
...   load_example_pair('example_5_26'),
...   load_example_pair('inf_conj'),
...   random_conjugate_infinite_pair()):
...     rho_i = psi_i.test_conjugate_to(phi_i)
...     print('Conjugate:', rho_i is not None, ' Conjugator works:', rho_i *␣
↪phi_i == psi_i * rho_i)
```

(continues on next page)

(continued from previous page)

```
Conjugate: True  Conjugator works: True
Conjugate: True  Conjugator works: True
Conjugate: True  Conjugator works: True
```

```
>>> f, g = (load_example('not_conjugate_' + c) for c in 'fg')
>>> f.is_conjugate_to(g)
False
```

See also:

This implements Algorithm 5.27 of the paper—see Section 5.4.

equivalence_data(*type_b*, *type_c*)

Let X be the quasi-normal basis for the current automorphism ψ. We can define an equivalence relation $\equiv$ on X by taking the non-transitive relation

$$x \equiv y \iff \exists k, \Gamma, \Delta : x\Gamma\psi^k = y\Delta$$

and allowing it to generate an equivalence relation. This method returns a pair *(roots, graph)* where

- *graph* is a `DiGraph()`
  - vertices are type B words in X
  - directed edges $x \to y$ correspond to the direct relation $x \equiv y$
  - the *graph* is a directed forest
- *roots* is a list of type B words in X
  - Each *root* is the root of a different tree in the forest *graph*.

This information allows us to (attempt to) compute the images of X under a conjugator given only the images of *roots*.

See also:

Definition 5.17 through to Lemma 5.20.

potential_image_endpoints(*self_type_b*)

Let `x` be a type B word with respect to the current automorphism. This returns a mapping which takes `x` and produces the set of words `w` which are endpoints of *other*-orbits which have the same characteristic as `x`.

See also:

The sets $\mathcal{R}_i$ of defintion 5.23.

find_power_conjugators(*other*, *cheat=False*)

Tests two infinite factors to see if they are power conjugate. Yields minimal solutions (a, b, ho).

```
>>> example_6_8_psi, example_6_8_phi = load_example_pair('example_6_8')
>>> solns = example_6_8_psi.find_power_conjugators(example_6_8_phi)
>>> for a, b, rho in solns:
...     print(a, b)
...     assert example_6_8_psi ** a * rho == rho * example_6_8_phi ** b
3 1
-3 -1
```

> **Warning:** If the *power_conjugacy_bounds()* are reasonable large (say > 30), this method could potentially take a long time!

See also:

Algorithm 6.11 of the paper.

test_power_conjugate_to(*other*, *cheat=False*)

Tests two infinite factors to see if they are power conjugate. If a solution exists, returns (a, b, ρ) such that $\rho^{-1}\psi^a\rho = \phi^b$. Otherwise returns None.

```
>>> result = example_6_8_psi.test_power_conjugate_to(example_6_8_phi)
>>> result is not None
True
>>> a, b, rho = result
>>> (example_6_8_psi ** a) * rho == rho * (example_6_8_phi ** b)
True
```

> **Warning:** If the *power_conjugacy_bounds()* are reasonable large (say > 30), this method could potentially take a long time!

See also:

Algorithm 6.11 of the paper.

power_conjugacy_bounds(*other*)

Compute the bounds $\hat{a}, \hat{b}$ on powers which could solve the power conjugacy problem.

```
>>> example_6_8_psi.power_conjugacy_bounds(example_6_8_phi)
(9, 1)
```

See also:

Proposition 6.6 and Corollary 6.7.

minimal_partition()

Let ψ be the current automorphism. This method partitions the characteristics M_ψ into cells $P_1 \sqcup \cdots \sqcup P_L$, where - The multipliers Γ all have the same *root()*, for all (m, Γ) in each P_i. - L is minimal with this property.

```
>>> def print_partition(p):
...     for root in sorted(p.keys(), reverse=True):
...             print(format(root), end = ':')
...             for power, mult, _ in p[root]:
...                     print(' ({}, {})'.format(power, format(mult)), end='')
...             print()
>>> print_partition(example_6_8_psi.minimal_partition())
a1: (-1, a1)
a2: (1, a2)
>>> print_partition(example_6_8_phi.minimal_partition())
a1: (-1, a1 a1 a1)
a2: (1, a2 a2 a2)
>>> example_6_9_psi, example_6_9_phi = load_example_pair('example_6_9')
>>> print_partition(example_6_9_phi.minimal_partition())
a1: (-2, a1)
a2: (1, a2)
```

Returns a dictionary of sets. The keys of this dictionary are the roots $\sqrt{\Gamma}$; the values are the cells P_i. An element of a cell looks like m, Γ, r where m, Γ is a characteristic and r is the root power corresponding to $\sqrt{\Gamma}$.

See also:

the discussion following Corollary 6.7.

static compute_bounds(*s_parts*, *o_parts*)

Computes the bounds $\hat{a}, \hat{b}$ (in terms of the partitions P, Q given by *s_parts* and *o_parts*) as in eqns (14) and (15) of the paper.

```
>>> def bounds(s, o):
...     P = s.minimal_partition()
...     Q = o.minimal_partition()
...     a_hat = InfiniteAut.compute_bounds(P, Q)
...     b_hat = InfiniteAut.compute_bounds(Q, P)
...     return a_hat, b_hat
>>> bounds(example_6_8_psi, example_6_8_phi)
(9, 1)
>>> bounds(example_6_9_psi, example_6_9_phi)
(1, 2)
```

3.9.3 Next steps

With these classes, the conjugacy and power conjugacy tests are implemented. The other important part of this package is the *examples* module.

3.10 Examples

This module provides a number of explicit examples for use in doctests. Additionally, functions to generate random automorphisms are provided.

3.10.1 Explict named examples

A list of named examples is loaded from the `.aut` files in the examples folder. This includes all the examples given in the paper, as well as others used to test the package. Use the following functions to load one of these examples.

thompson.examples.**available_examples**()

Returns an iterator yielding the names of the examples that are provided with the package. (Note that *the full list is provided* in this documentation.)

```
>>> list(available_examples())[:4]
['alphabet_size_two', 'arity_four', 'arity_three_order_inf', 'bleak_alpha']
```

thompson.examples.**load_example**(*name*)

Loads the example with the given *name* from disk. A corresponding *Automorphism* instance is created and returned. The results are cached, so call this method as often as you like.

thompson.examples.**load_example_pair**(*name*)

Loads a pair of examples, `*name*_psi` and `*name*_phi`.

Return type a 2-tuple of automorphisms.

thompson.examples.**load_all_examples**()

Loads (and processes) **all** examples provided by the package. Returns a dictionary whose keys are the example names and whose values are the loaded automorphisms.

Note: To discourage use of this function, it is not available when importing * from thompson.examples. Instead it must be explicitly imported with from thompson.examples import load_all_examples.

Warning: Some of the examples are slow to process, so calling this could take a long time.

thompson.examples.**standard_generator**(*n=0*)

Produces the standard generator X_n of F as described in *[CFP96]*. For instance, X_0 is the following:

```
>>> print(standard_generator())
InfiniteAut: V(2, 1) -> V(2, 1) specified by 3 generators (after expansion and␣
↪reduction).
x1 a1    -> x1 a1 a1
x1 a2 a1 -> x1 a1 a2
x1 a2 a2 -> x1 a2
```

For $n > 0$ the element X_n is a *Mixed automorphism*, consisting of a large fixed part and a smaller part which looks like X_0.

```
>>> from random import randint
>>> n = randint(1, 20)
>>> x_n = standard_generator(n)
>>> type(x_n)
<class 'thompson.mixed.MixedAut'>
```

The X_n generate F; in fact just X_0 and X_1 are sufficient, due to the relation $X_k^{-1} X_n X_k = X_{n+1}$ for $k < n$. See *[CFP96]* for more details.

```
>>> x_k = standard_generator(randint(0, n-1))
>>> x_k * x_n * ~x_k == standard_generator(n+1) #operation is the other way round␣
↪in Python
True
```

Todo: Add the named generators A, B, C of Thompson's T and V. Analogues for general G_n,r?

3.10.2 List of named examples

Note that some automorphisms have more than one name—they will appear once in this list for every alias.

alphabet_size_two See the forest diagram and function plot.

A toy example to test that the program works in $V_{3,2}$.

arity_four See the forest diagram and function plot.

Another example for sanity checking. This is an automorphism of $V_{4,1}$.

arity_three_order_inf See the forest diagram and function plot.

A purely *infinite* automorphism of $V_{3,1}$.

bleak_alpha See the `forest` diagram and function `plot`.

The element α of V used by Bleak *et al [BBG11]* to illustrate their train tracks and flow graphs.

bleak_klein_alpha See the `forest` diagram and function `plot`.

The example in *[BBG11]* of an element whose centraliser contains the Klein four group $\mathbb{Z}_{\not\equiv} \oplus \mathbb{Z}_{\not\equiv}$ as a subgroup. We have represented it as an element of $G_{2,8}$ rather than $G_{2,1}$.

bleak_klein_beta See the `forest` diagram and function `plot`.

The example β in *[BBG11]* of an element generating a Klein four group as part of the centraliser of `bleak_klein_alpha`.

bleak_klein_gamma See the `forest` diagram and function `plot`.

The example γ in *[BBG11]* of an element generating a Klein four group as part of the centraliser of `bleak_klein_alpha`.

commutator See the `forest` diagram and function `plot`.

An element of the commutator subgroup of F. It has one bump and two fixed intervals.

cyclic_order_six See the `forest` diagram and function `plot`.

An introductory example drawn on the board by NB. It is purely periodic of order six.

example_4_1 See the `forest` diagram and function `plot`.

This example is used in the paper to illustrate the meaning of a semi-normal form (Example 4.11) and of a tree pair diagram (Example 4.1).

example_4_12 See the `forest` diagram and function `plot`.

This example is used in the paper to illustrate the meaning of a semi-normal form. This example needs to be expanded from its minimal representation to find a semi-normal (in fact quasi-normal) form.

example_4_5 See the `forest` diagram and function `plot`.

This example is used to illustrate the different types of *components/orbits* in the paper.

example_5_12_phi See the `forest` diagram and function `plot`.

The example used in the paper to demonstrate the *periodic conjugacy test*.

example_5_12_psi See the `forest` diagram and function `plot`.

The example used in the paper to demonstrate the *periodic conjugacy test*.

example_5_15 See the `forest` diagram and function `plot`.

This example is used in the paper to illustrate the process of finding the *quasi-normal basis*.

example_5_26_psi See the `forest` diagram and function `plot`.

The example used in the paper to demonstrate the *infinite conjugacy test*.

example_5_3 See the `forest` diagram and function `plot`.

This example is used to demonstrate how we can break down an automorphism into its *free_factors()*.

example_5_9 See the `forest` diagram and function `plot`.

This example is used in the paper to illustrate the definition of *cycle types*.

example_6_2 See the `forest` diagram and function `plot`.

Example 6.2 of the paper, used to illustrate lemma 6.1.

example_6_8_phi See the `forest` diagram and function `plot`.

A purely infinite automorphism which illustrates the bounds $\hat{a}, \hat{b}$ for power conjugacy.

f_thirds_fixed See the `forest` diagram and function `plot`.

An element of F whose fixed point set is exactly $\{0, 1/3, 2/3, 1\}$.

first_pond_example_phi See the `forest` diagram and function `plot`.

The example of a pond orbit we found by a random search. Its banks are $x\alpha_1^4\alpha_2$ and $x\alpha_1\alpha_2^2$.

first_pond_example_psi See the `forest` diagram and function `plot`.

The element which was conjugate to `first_pond_example_phi`.

four_minimal_revealing_pairs See the `forest` diagram and function `plot`.

I think this automorphism has 4 minimal revealing pairs. It has an orbit of characteristic (-4, a1^4) spread apart over four components of $A \setminus B$. Brin's algorithm would have you add three of these components to the tree, and I think the component you DON'T add is the one that determines the revealing pair.

inf_conj_phi See the `forest` diagram and function `plot`.

An example used to debug the *infinite conjugacy test*.

inf_conj_psi See the `forest` diagram and function `plot`.

An example used to debug the *infinite conjugacy test*.

mixed_pconj_phi See the `forest` diagram and function `plot`.

An example of mixed power conjugacy found by random search. Should have $\psi^{-4} \sim \phi^2$.

mixed_pconj_psi See the `forest` diagram and function `plot`.

An example of mixed power conjugacy found by random search. Should have $\psi^{-4} \sim \phi^2$.

mixed_plines See the `forest` diagram and function `plot`.

An element of T with rotation number $1/2$. The square of this element acts like x_0^2 on the first and third quarters, and acts like the identity elsewhere. Thus it has two periodic flow lines and two pointwise periodic lines. The same is true of the square of mixed_plines_2, but even though it does not act as the identity on the second and fourth quarters.

mixed_plines_2 See the `forest` diagram and function `plot`.

An element of T with rotation number $1/2$ and no fixed intervals. The square of this element acts like x_0^2 on the first and third quarters, and acts like the identity elsewhere. Thus it has two periodic flow lines and two pointwise periodic lines. The same is true of the square of mixed_plines, which acts as the identity on the second and fourth quarters.

multiple_classes See the `forest` diagram and function `plot`.

A purely infinite automorphism for which there are two equivalence classes defined on X.

multiple_classes_smaller See the `forest` diagram and function `plot`.

A purely infinite automorphism for which there are two equivalence classes defined on X. It was more straightforward to see that this automorphism had two equivalence classes.

nathan1_example See the `forest` diagram and function `plot`.

One of Nathan's examples. This is *almost*, but not quite conjugate to nathan_pond_example. Nathan had this to say: *"I untwisted one of the infinite orbits after a conjugation and so they definitely should not be conjugate."*

nathan_pond_example See the `forest` diagram and function `plot`.

Nathan's example of an automorphism containing a pond. This has a simpler tree pair than `first_pond_example_phi`.

no_minimal_revealing See the `forest` diagram and function `plot`.

An example from section 3.6 of *[SD10]*. This is an automorphism $f \in G_{2,1}$ for which there is no minimal revealing pair from which every other revaling pair can be obtained via rollings.

non_dyadic_fixed_point See the `forest` diagram and function `plot`.

In their paper introducing strand diagrams, Belk and Matucci provide an example of an element of F which has a non-dyadic fixed point. This is a slight simplification of that example.

non_revealing See the `forest` diagram and function `plot`.

A purely infinite automorphism for which - The minimal representation does NOT correspond to a revealing pair; and - The quasinormal basis does NOT correspond to a revealing pair.

not_conjugate_f See the `forest` diagram and function `plot`.

An example of an element which isn't conjugate to `not_conjugate_g`, even though it comes quite close to being so!

not_conjugate_g See the `forest` diagram and function `plot`.

An element which is not conjugate to `not_conjugate_f`, even though it comes quite close to being so!

olga_f See the `forest` diagram and function `plot`.

An automorphism described in example 5 of *[SD10]*. Nathan had this to say: "*olga_f is conjugate to olga_g, via a rho that can be found using your program or olga_h. Note that characteristics are not the same for two of the semi-infinite orbits.*"

olga_g See the `forest` diagram and function `plot`.

An automorphism described in example 5 of *[SD10]*. Nathan had this to say: "*olga_f is conjugate to olga_g, via a rho that can be found using your program or olga_h. Note that characteristics are not the same for two of the semi-infinite orbits.*"

olga_gdash See the `forest` diagram and function `plot`.

An automorphism described in example 5 of *[SD10]*. The paper claims that `olga_f` and `olga_gdash` are not conjugate.

olga_h See the `forest` diagram and function `plot`.

An automorphism described in example 5 of *[SD10]*. Nathan had this to say: "*olga_f is conjugate to olga_g, via a rho that can be found using your program or olga_h. Note that characteristics are not the same for two of the semi-infinite orbits.*"

periodic_QNB_206 See the `forest` diagram and function `plot`.

This purely periodic automorphism (again found by random search) has a quasinormal basis of size 206.

periodic_QNB_344 See the `forest` diagram and function `plot`.

This purely periodic automorphism (again found by random search) has a quasinormal basis of size 344.

pond_width_4 See the `forest` diagram and function `plot`.

An example found by random search of a pond orbit of width 4. Prior to this we had only seen ponds of width 1 or 2.

power_smaller_QNB See the `forest` diagram and function `plot`.

The square of this automorphism appears to have a QNB above the QNB of the original automorphism.

rotation_number_one_third See the `forest` diagram and function `plot`.

An non-periodic element of T_{2,3} which should have rotation number 1/3.

scott_free_alpha See the `forest` diagram and function `plot`.

The element alpha in the appendix of *[Scott]* which generates a free group of rank 2 with beta, also from the appendix.

scott_free_beta See the `forest` diagram and function `plot`.

The element beta in the appendix of *[Scott]* which generates a free group of rank 2 with alpha, also from the appendix.

semi_inf_c See the `forest` diagram and function `plot`.

An example of an automorphism whose *quasi-normal basis* X contains an element $x\alpha_2\alpha_1$ which belongs to a semi-infinite X-component which is not characteristic.

t_generator_c See the `forest` diagram and function `plot`.

This is the element C used by Cannon, Floyd and Parry to generate Thompson's group T.

thirds_fixed See the `forest` diagram and function `plot`.

This element of T has only two fixed points: 1/3 and 2/3; note that these are non dyadic!

two_flow_components_d See the `forest` diagram and function `plot`.

An example which found a bug in the infinite conjugacy test. This automorphism contains two flow components which are identical in all but their location.

two_flow_components_e See the `forest` diagram and function `plot`.

An example which found a bug in the infinite conjugacy test. This automorphism is a conjugated version of `two_flow_components_d`.

v_generator_pi See the `forest` diagram and function `plot`.

This is the generator π_0 used by Cannon, Floyd and Parry to generate V.

v_revealing_test See the `forest` diagram and function `plot`.

A short example used to test the is_revealing method

3.10.3 Randomly generated examples

Words

thompson.examples.**random_signature**()

Randomly generates a *Signature* (n, r) for use in the functions below. The values of n and r are chosen (uniformly) at random from $n \in \{2, 3, 4\}$ and $r \in \{1, 2, 3, 4, 5\}$, respectively.

Note: This function is used to select a random signature when no *signature* argument is provided to the following random functions.

---

thompson.examples.**random_simple_word**(*signature=None*)

Randomly generates a *simple Word* belonging to the algebra with the given *signature*. The word consists of an x_i followed by 0–15 descendant operators α_j. The length and the index of each α_j is chosen (uniformly) at random.

```
>>> random_simple_word().is_simple()
True
```

thompson.examples.**random_basis**(*signature=None*, *num_expansions=None*, **args*, ***kwargs*)

Randomly generates a basis for the algebra with the given *signature* (n, r). The basis is generated by expanding the *standard_basis()* *num_expansions* times. The expansion point is chosen (uniformly) at random each time. If *num_expansions* is not provided, a value from $\{1, 2, 3, 4, 5\}$ is chosen (uniformly) at random.

```
>>> random_basis().is_basis()
True
```

The *cls* argument specifies which class the generated basis should have; this should be a subclass of *Generators*. At the moment, this is only used internally and experimentally.

Note: This does not generate *bases* uniformly at random. For instance, take $V_{n,r} = V_{2,1}$ and let *num_expansions* = *3*. The first expansion always gives the basis $[x\alpha_1, x\alpha_2]$. Expanding this twice produces give six bases, one of which appears twice. (To see this, enumerate the rooted binary trees with four leaves.)

Automorphisms

thompson.examples.**random_automorphism**(*signature=None*, *num_expansions=None*, **args*, ***kwargs*)

Randomly generates an automorphism for the algebra with the given signature. Two bases *domain* and *range* are generated by *random_basis()*. Then the *range* is manipulated according to the group we're interesting in. For `group == 'V'`, we randomly shuffle the *range*; for `group == 'T'`, we cyclicly permute the *range*; for `group == 'F'` we do nothing.

```
>>> random_automorphism(group='T').cycles_order()
True
>>> random_automorphism(group='F').preserves_order()
True
```

An automorphism is returned which maps the elements of *domain* to those of *range* in whichever order results.

Note: The bases may be reduced when the automorphism is created (if they contain redundancy).

---

thompson.examples.**random_periodic_automorphism**(*signature=None*, *num_expansions=None*, **args*, ***kwargs*)

Randomly generates an automorphism for the algebra with the given signature. A basis *domain* is generated by *random_basis()*. A copy *range* is made of *domain*, and is then randomly shuffled. An automorphism is returned which maps the elements of *domain* to those of *range* in order.

Note: The bases may be reduced when the automorphism is created (if they contain redundancy.

---

thompson.examples.**random_infinite_automorphism**(*signature=None*, *num_expansions=None*, **args*, ***kwargs*)

Randomly generates an infinite automorphism—either by chance, or by extracting an infinite *free_factor()*.

Todo: The implementation is inefficient: just make auts at random until you find an infinite one. Fix this!

---

thompson.examples.**random_conjugate_pair**(*signature=None*, *num_expansions=None*, **args*, ***kwargs*)

Calls *random_automorphism()* to create two automorphisms ψ and ρ. Returns the pair $(\psi, \rho^{-1}\psi\rho)$, which are conjugate by definition.

thompson.examples.**random_conjugate_periodic_pair**(*signature=None*, *num_expansions=None*, **args*, ***kwargs*)

The same as *random_conjugate_pair()*, except ψ is chosen to be a *random_periodic_automorphism()*.

thompson.examples.**random_conjugate_infinite_pair**(*signature=None*, *num_expansions=None*, **args*, ***kwargs*)

The same as *random_conjugate_pair()*, except ψ is chosen to be a *random_infinite_automorphism()*.

thompson.examples.**random_power_conjugate_pair**(*signature=None*, *num_expansions=None*, **args*, ***kwargs*)

Generates ψ, ρ using *random_automorphism()* and random powers a, b. Returns the tuple $(\Psi, \Phi) = (\psi^b, \rho^{-1}\psi^a\rho)$. Note that Ψ^a is conjugate to Φ^b via ρ.

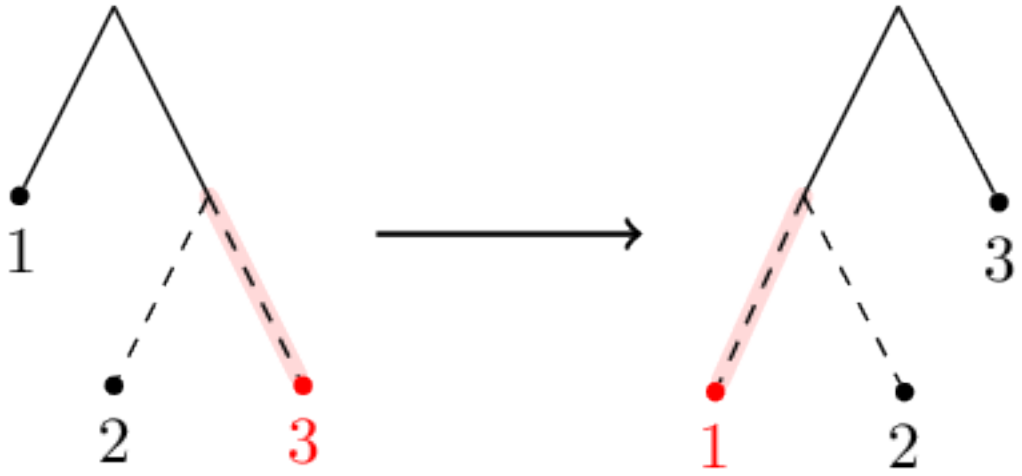
3.11 Drawing automorphisms

It can be helpful to see automorphisms rather than just read a description of them. To that end we introduce functions for rendering automorphisms as tree pair diagrams. The output looks best for automorphisms with a small arity (2–5) and a reasonable number of leaves.

3.11.1 Examples

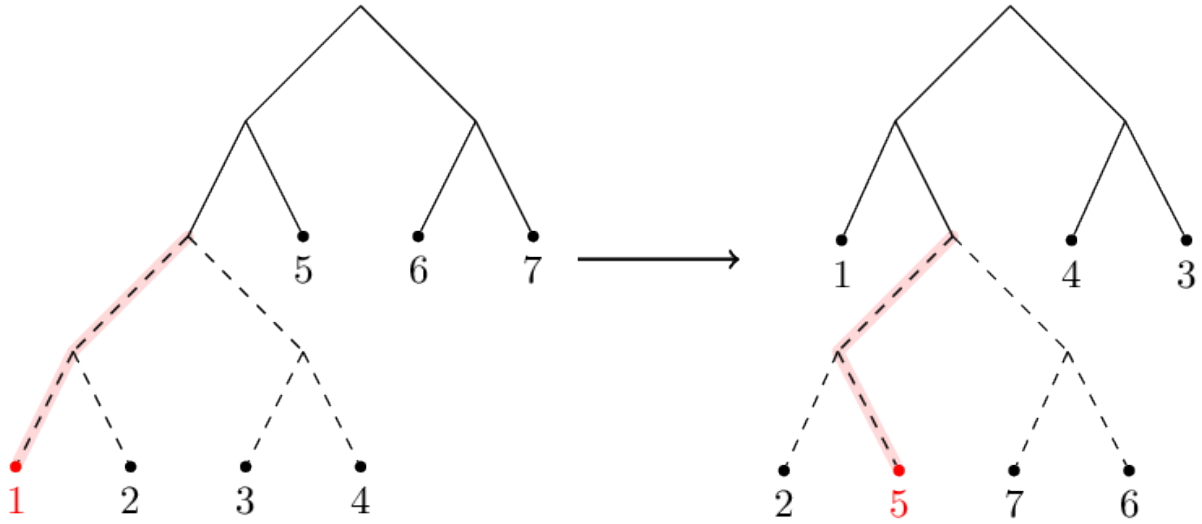
First we take a well-behaved automorphism. The solid carets are those belonging to both the domain and range trees. All other carets are dotted. Some edges are highlighted in red—these correspond to the repellers and attractors discussed by *[SD10]*.

```
>>> from thompson import *
>>> x0 = standard_generator(0)
>>> plot(x0)
>>> forest(x0)
```



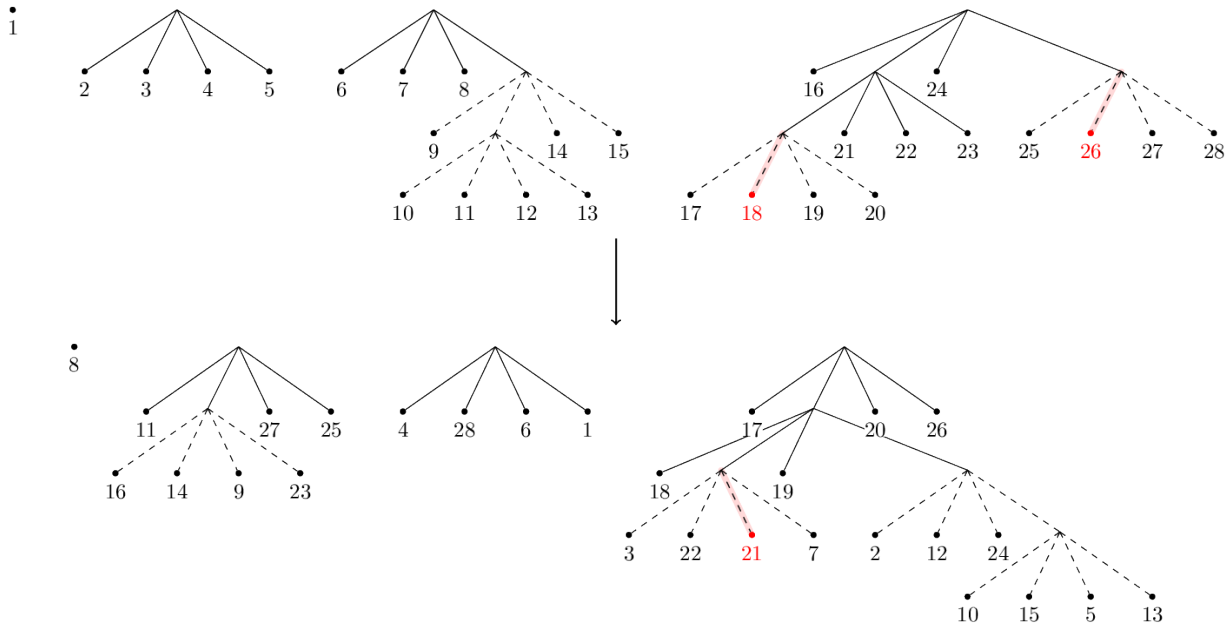
Discontinuities are fine too. Here's `example_4_17` for instance.

```
>>> pond = load_example('example_4_17')
>>> plot(pond, discontinuities=True)
>>> forest(pond)
```



Let's aim for something more chaotic. This example is complicated enough that the drawings don't really give us much insight into the automorphism.

```
>>> random = random_automorphism() #different every time!
>>> plot(random)
>>> forest(random, horiz=False)
```



3.11.2 Drawing functions

thompson.drawing.**display_file**(*filepath*, *format=None*, *scale=1.0*, *verbose=False*)

Display the image at *filepath* to the user. This function behaves differently, depending on whether or not we execute it in a Jupyter notebook.

If we are **not** in a notebook, this function opens the given image using the operating system's default application for that file.

If we **are** in a notebook, this returns an IPython object corresponding to the given image. If this object is the last expression of a notebook code block, the image will be displayed. The image is handled differently depending on its *format*, which must be specified when the function is called in a notebook. Only the formats *'svg'* and *'pdf'* are accepted.

Note: PDF files are displayed in a notebook as a rendered PNG. The conversion is made using the `convert` program provided by ImageMagick, which must be available on the PATH for this function to work with PDF files. The optional *scale* argument can be used to control how big the rendered PNG is.

Todo: use a Python binding to ImageMagick rather than just shelling out?

---

thompson.drawing.**plot**(**auts*, *dest=None*, *display=True*, *diagonal=False*, *endpoints=False*, *scale=1*)

Plots the given `automorphism` s as a function $[0,1] \to [0,1]$. The image is rendered as an SVG using *svgwrite*.

Parameters

- **dest** (`str`) – the destination filepath to save the SVG to. If *None*, the SVG is saved to a temporary file location.
- **display** (`bool`) – if True, automatically call `display_file()` to display the SVG to the user. Otherwise does nothing.

- **diagonal** (*bool*) – if True, draws the diagonal to highlight fixed points of *aut*.
- **endpoints** (*bool*) – if True, open and closed endpoints are drawn on the plot to emphasise any discontinuities that *aut* has.

Returns the filepath where the SVG was saved. If *dest* is *None* this is a temporary file; otherwise the return value is simply *dest*.

thompson.drawing.**forest**(*aut*, *jobname=None*, *display=True*, *scale=1*, ***kwargs*)

Draws the given *Automorphism* as a forest-pair diagram. The image is rendered as a PDF using the *tikz* graph drawing libraries and *lualatex*.

Parameters

- **jobname** (*str*) – the destination filepath to save the PDF to. A file extension should **not** be provided. If *None*, the PDF is saved to a temporary file location.
- **display** (*bool*) – if True, automatically call *display_file()* to display the PDF to the user. Otherwise does nothing.
- **scale** (*float*) – In a Jupyter notebook, this controls the size of the rendered PNG image. See the note in *display_file()*.

Returns the filepath where the PDF was saved. If *dest* is *None* this is a temporary file; otherwise the return value is simply *jobname* + *'.pdf'*.

Note: The graph drawing is done via a TikZ and LaTeX. The source file is compiled using *lualatex*, which must be available on the PATH for this function to work.

---

thompson.drawing.**flow**(*aut*, *jobname=None*, *display=True*)

thompson.drawing.**forest_code**(*aut*, *name=''*, *domain='wrt QNB'*, *include_styles=True*, *horiz=True*, *LTR=True*, *standalone=True*, *draw_revealing=True*)

Generate TikZ code for representing an automorphism *aut* as a forest pair diagram. The code is written to *dest*, which must be a writeable file-like object in text mode.

Parameters

- **name** (*str*) – The label used for the arrow between domain and range forests. This is passed directly to TeX, so you can include mathematics by delimiting it with dollars. Note that backslashes are treated specially in Python unless you use a *raw string*, which is preceeded with an `r`. For instance, try `name=r'\gamma_1`
- **domain** (Generators) – By default, we use the *minimal expansion* of the *quasi-normal basis* as the leaves of the domain forest. This can be overridden by providing a *domain* argument.
- **include_styles** (*bool*) – Should the styling commands be added to this document?
- **horiz** (*bool*) – If True, place the range forest to the right of the domain. Otherwise, place it below.
- **LTR** (*bool*) – if True and if horiz is True, draw the domain tree to the left of the range tree. If horiz is True but LTR is false, draw the domain tree to the right of the range tree.
- **standalone** (*bool*) – If True, create a standalone LaTeX file. Otherwise just create TikZ code.
- **draw_revealing** (*bool*) – Should attractor/repeller paths be highlighted in red?

3.12 References

The main reference is to the paper describing theory behind these algorithms.

3.12.1 From the paper

3.12.2 Other references

3.13 Todo list

Note: There are also many `#TODO` comments scattered throughout the python files.

Warning: Auto-generated page! This does not look pretty.

Todo: Make this available under an open-source license.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/latest/docs/index.rst, line 38.)

Todo: This could be made more efficient for large values of *power* by using knowledge of the component containing *key*.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/latest/thompson/automorphism.py:docstring of thompson.automorphism.Automorphism.repeated_image, line 18.)

Todo: This should be renamed to component_type.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/latest/thompson/automorphism.py:docstring of thompson.automorphism.Automorphism.orbit_type, line 91.)

Todo: This method is badly named; something like `test_revealed_by(basis)` would be better.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/latest/thompson/automorphism.py:docstring of thompson.automorphism.Automorphism.test_revealing, line 21.)

Todo: This method is badly named; something like `is_revealed_by(basis)` would be better.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/latest/thompson/automorphism.py:docstring of thompson.automorphism.Automorphism.is_revealing, line 21.)

Todo: doctests

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/latest/thompson/automorphism.py:docstring of thompson.automorphism.Automorphism.centralise_period, line 5.)

Todo: use a Python binding to ImageMagick rather than just shelling out?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/latest/thompson/drawing/__init__.py:docstring of thompson.drawing.display_file, line 14.)

Todo: Add the named generators A, B, C of Thompson's T and V. Analogues for general G_n,r?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/latest/docs/thompson.examples.rst, line 19.)

Todo: The implementation is inefficient: just make auts at random until you find an infinite one. Fix this!

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/latest/thompson/examples/random.py:docstring of thompson.examples.random_infinite_automorphism, line 3.)

Todo: Maybe this should be 0 to order if the identity is permitted and 1 to order otherwise?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/latest/thompson/periodic.py:docstring of thompson.periodic.PeriodicAut.power_conjugacy_bounds, line 5.)

Todo: Should use a different attribute (.comment?) rather than __doc__.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/latest/thompson/homomorphism.py:docstring of thompson.homomorphism.Homomorphism.from_file, line 20.)

Todo: More convenient ways of inputting words, e.g. `x a1^3` instead of `x a1 a1 a1`. Use of unicode characters α and λ?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/latest/thompson/word.py:docstring of thompson.word.from_string, line 36.)

Todo: I think this is a total order—try to prove this. I based the idea on *[Zaks]* (section 2, definition 1) which describes a total order on *k*-ary trees. On another note, isn't this similar to shortlex order?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/latest/thompson/word.py:docstring of thompson.word.Word.__lt__, line 53.)

3.14 Open subsets of the Cantor set

class thompson.cantorsubset.**CantorSubset**(*signature*, *generators=None*)

A subclass of *Generators* specifically designed to represent an open subset of the Cantor set $\mathcal{C}$. We store this as a (typically sorted) list of *Generators* $u_1, \ldots, u_n$. This stands for the set of all points $u_i\Gamma$ where $\Gamma \in \{0,1\}^{\mathbb{N}}$.

is_entire_Cantor_set()

is_empty()

status()

contract()

Contracts the current generating set as much as possible (without using words involving a λ). The set should be sorted before using this method.

```
>>> basis = random_basis(cls=CantorSubset)
>>> basis.contract()
>>> basis == CantorSubset.standard_basis(basis.signature)
True
>>> basis = CantorSubset((2,1), '01011 01010'.split())
>>> basis.contract()
>>> print(basis)
[0101]
```

simplify()

Simplifies the current Cantor subset to a normal form. We do this in three steps. Firstly we expand any *nonsimple Words*. Second, we *contract()* as much as possible (without using words involving a λ). Last, we check to see if the set contains any pairs $u, u\Gamma$ and remove the latter.

```
>>> X = CantorSubset((2,1), "0 11 10 00 111 1101 11110".split())
>>> X.simplify(); print(X)
[<entire Cantor set>]
>>> X = CantorSubset((2,1), ["1"])
>>> X.simplify(); print(X)
[1]
```

__str__()

We use a notation introduced to us by Bleak: square brackets around a word stand for "the Cantor set underneath" its argument. We use the *format_cantor()* function to display the elements of the generating set.

```
>>> print(CantorSubset((2, 1)))
[]
>>> print(CantorSubset((2, 1), ["x"]))
[<entire Cantor set>]
>>> print(CantorSubset((2, 1), ["x a1"]))
[0]
>>> S = CantorSubset((2, 1), ["x a1", "x a1 a1", "x a2 a1 a1", "x a2 a1 a2"])
>>> print(S)
[0, 00, 100, 101]
>>> S.simplify(); print(S)
[0, 10]
```

__and__(*other*)

Computes the intersection of Cantor subsets. We assume the list of words is sorted before calling this function.

```
>>> A = CantorSubset((2, 1), ["00"])
>>> B = CantorSubset((2, 1), "01 111".split())
>>> C = A & B
>>> print(A, B, C)
[00] [01, 111] []
>>> D = CantorSubset((2, 1), ["11"])
>>> print(D, B, D & B)
[11] [01, 111] [111]
```

__invert__()

The complement. again it only works on a sorted list of leaves.

```
>>> X = CantorSubset((2, 1), "01 1010 11".split())
>>> print(~X)
[00, 100, 1011]
>>> print(~CantorSubset((2, 1), []))
[<entire Cantor set>]
>>> print(~CantorSubset((2, 1), ["x1"]))
[]
```

```
>>> X = random_generators(cls=CantorSubset, signature=(2,1))
>>> X.sort();
>>> comp = ~X
>>> print(X & comp)
[]
>>> X.extend(comp); X.simplify()
>>> print(X)
[<entire Cantor set>]
```

thompson.cantorsubset.**detailed_comparison**(*self*, *other*)

Returns a tuple *(subword, comparison)* of two Booleans. Either one word is a subword of the other or not. In the former case *subword* is `True`, otherwise *subword* is `False`.

In both cases we can decide if *self* is lexicographically smaller than *other*. If so, *comparison* is `-1`. If they are equal, *comparison* is `0`; if *self* is larger than *other* then *comparison* is `1`.

```
>>> for thing in ["0 00", "001 00", "101 101", "0 1", "11 00"]:
...     thing = thing.split()
...     s = Word(thing[0], (2, 1))
...     o = Word(thing[1], (2, 1))
...     print(*detailed_comparison(s, o))
...
True -1
True 1
True 0
False -1
False 1
```

CHAPTER 4

Indices

- genindex

Bibliography

[BDR] N. Barker, A. J. Duncan, and D. M. Robertson, "The power conjugacy problem in Higman-Thompson groups", *International Journal of Algebra and Computation* 26, Issue 2 (2016). Preprint available at arXiv:1503.01032 [math.GR].

[AS74] M. Anshel and P. Stebe, "The solvability of the conjugacy problem for certain HNN groups", *Bull. Amer. Math. Soc.* , **80** (2) (1974) 266–270.

[B87] K. S. Brown, "Finiteness properties of groups", Journal of Pure and Applied Algebra, **44** (1987) 45-75.

[BBG11] C. Bleak, H. Bowman, A. Gordon, G. Graham, J. Hughes, F. Matucci and J. Sapir, "Centralizers in R.Thompson's group V_n", *Groups, Geometry and Dynamics* **7** , No. 4 (2013), 821–865.

[BCR] J. Burillo, S. Cleary and C. E. R"over, "Obstructions for subgroups of Thompson's group V", arxiv.org/abs/1402.3860

[BK08] V. N. Bezverkhnii, A.N. Kuznetsova, "Solvability of the power conjugacy problem for words in Artin groups of extra large type", *Chebyshevskii Sb.* **9** (1) (2008) 50–68.

[BM07] J. M. Belk and F. Matucci, "Conjugacy and dynamics in Thompson's groups", *Geom. Dedicata* **169** (1) (2014) 239–261.

[BMOV] O. Bogopolski, A. Martino, O. Maslakova and E. Ventura, "The conjugacy problem is solvable in free-by-cyclic groups", *Bulletin of the London Mathematical Society* , **38** , (10) (2006) 787–794.

[Bar] N. Barker, "Topics in Algebra: The Higman-Thompson Group $G_{2,1}$ and Beauville p-groups", *Thesis, Newcastle University* (2014)

[Bez14] N. V. Bezverhnii, "Ring Diagrams with Periodic Labels and Power Conjugacy Problem in Groups with Small Cancellation Conditions C (3) -T (6)", *Science and Education of the Bauman MSTU* , **14** (11) (2014).

[Brin04] M. G. Brin, "Higher dimensional Thompson groups", *Geom. Dedicata* , **108** (2004) 163–192.

[CFP96] J. W. Cannon, W.J. Floyd and W. R. Parry, "Introductory notes on Richard Thompson's groups", *Enseign. Math.* , (2) **42** (3–4) (1996) 215–256.

[Cohn81] P. M. Cohn, "Universal Algebra". Mathematics and its Applications, 6, D. Reidel Pub. Company, (1981).

[Cohn91] P. M. Cohn, "Algebra, Volume 3". J. Wiley, (1991).

[Com77] L. P. J. Comerford, A note on power-conjugacy, *Houston J. Math.* 3 (1977), no. 3, 337—341.

[DMP14] W. Dicks, C. Martinez-Pérez, "Isomorphisms of Brin-Higman-Thompson groups", *Israel Journal of Mathematics* , **199** (2014), 189–218.

[Fes14] A. V. Fesenko, "Vulnerability of Cryptographic Primitives Based on the Power Conjugacy Search Problem in Quantum Computing", *Cybernetics and Systems Analysis* , **50** (5) (2014) 815–816.

[Hig74] G. Higman, "Finitely presented infinite simple groups", *Notes on Pure Mathematics* , Vol. 8 (1974).

[JT61] B. Jónsson and A. Tarski, "On two properties of free algebras", *Math. Scand.* , **9** (1961) 95–101.

[KA09] D. Kahrobaei and M. Anshel, "Decision and Search in Non-Abelian Cramer-Shoup Public Key Cryptosystem", *Groups-Complexity-Cryptology* , **1** (2) (2009) 217-–225.

[LM71] S. Lipschutz and C.F. Miller, "Groups with certain solvable and unsolvable decision problems", *Comm. Pure Appl. Math.* , **24** (1971) 7–15.

[Lot83] M. Lothaire, "Combinatorics on Words", Addison-Wesley, Advanced Book Program, World Science Division, (1983).

[MPN13] C. Martinez-Perez, B. Nucinkis, "Bredon cohomological finiteness conditions for generalisations of Thompson's groups", *Groups Geom. Dyn.* **7** (4) (2013) 931–959.

[MT73] R. McKenzie and R. J. Thompson, "An elementary construction of unsolvable word problems in group theory", Word problems: decision problems and the Burnside problem in group theory, Studies in Logic and the Foundations of Math., 71, pp. 457–478. North-Holland, Amsterdam, (1973).

[Pa11] E. Pardo, "The isomorphism problem for Higman-Thompson groups", *Journal of Algebra* , **344** (2011), 172–183.

[Pr08] S. J. Pride, "On the residual finiteness and other properties of (relative) one-relator groups", *Proc. Amer. Math. Soc.* **136** (2) (2008) 377–386.

[R15] D. M. Robertson, "`thompson`: a package for Python 3.3+ to work with elements of the Higman-Thompson groups $G_{n,r}$". Source code available from https://github.com/DMRobertson/thompsons_v and documentation available from http://thompsons-v.readthedocs.org/.

[SD10] O. P. Salazar-Diaz, "Thompson's group V from a dynamical viewpoint", *Internat. J. Algebra Comput.* , 1, 39–70, 20, (2010).

[Tho] R. J. Thompson, unpublished notes. http://www.math.binghamton.edu/matt/thompson/index.html

[Kogan] R. Kogan, "nVTrees Applet" (2008). Accessed 17th September, 2014. Source code available on GitHub.

[Scott] E. Scott, "A finitely presented simple group with unsolvable conjugacy problem, *Journal of Algebra* 90, Issue 2, pp. 333–353 (1984).

[Zaks] S. Zaks, "Lexicographic generation of ordered trees", *Theoretical Computer Science* 10: 63–82 (1980).

Python Module Index

t

Index

Symbols

A

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V

W