Free group automorphisms and train-tracks with Sage
User's Guide

Thierry Coulbois

August 20, 2015

1 Introduction

The Train-track package was first written by Thierry Coulbois and received contributions by Matt Clay and others.

It is primarily intended to implement the computation of a train-track representative for automorphisms of free groups as introduced by Mladen Bestvina and Mark Feighn [**?**].

Sage is based on Python. This is an object oriented language: the **postfix convention** is used. For instance `phi.train-track()` applies the method `train-track()` to the object `phi`. Note that method is the object oriented linguo for what mathematicians call "function".

You can always ask for **automatic completion** and **help** by using the TAB key:

1. Hitting the TAB key after a letter offers all possible completions known to sage.
2. Hitting the TAB key after a dot shows all methods that can be applied to that object.
3. Hitting the TAB key after an opening parenthesis gives help on how this method should be used.

Most methods have **verbose options** to display intermediate computations. They are turned off by default, but you can supply a `verbose=True` option or any non-negative number to get extra details.

The main documentation for using this package is inline help and automatically created documentation. This User guide is only intended for beginners and general structure.

2 Installation and files

To use this package you first need a recent ($\geq$ 6.3) distribution of Sage. Then you need to download the files at

3 Free groups and automorphisms

3.1 Creating free groups

Probably you first need to create a free group. It can be specified by its rank or a list of letters. You can also first create the alphabet.

```
sage: F=FreeGroup(3); F
   Free group over ['a', 'b', 'c']
sage: F=FreeGroup(['x0','x1','x3','x4']); F
   Free group over ['x0', 'x1', 'x3', 'x4']
sage: A=AlphabetWithInverses(5,type='a0')
sage: F=F(A); F
   Free group over ['a0', 'a1', 'a2', 'a3', 'a4']
```

You can declare anything to be a letter, but beware that if letters are not single ascii characters (like 'x0'), you will need to be careful while going from Strings to Words.

3.2 Free group elements

Free group elements are words. They are created by

```
sage: F=FreeGroup(3)
sage: F('abA')
   word: abA
sage: w=F('abAaab'); w
   word: abAaab
sage: F.reduce(w)
   word: abab
```

Note that they are not reduced by default.

Words can be multiplied and inverted easily:

```
sage: w=F('abA')
sage: w*w
   word: abbA
sage: w.inverse()
   word: aBA
sage: w**5
   word: abbbbbA
```

Warning: be careful when the free group alphabet is not made of ascii letters:

```
sage: A=AlphabetWithInverses(3,type='x0')
sage: F=FreeGroup(A)
sage: ws='x0X0x1'
sage: w=F(ws); w
word: 'x0X0x1'
sage: F.reduce(w)
KeyError: 'x'
sage: w=F(['x0','X0','x1']); w
word: x0,X0,x1
sage: F.reduce(w)
word: x1
```

3.3 Free group automorphisms

The parsing of free group automorphisms relies on that of substitutions. Most of what you might expect should correctly create a free group automorphism:

```
sage: phi=FreeGroupMorphism('a->ab,b->a'); phi
Automorphism of the Free group over ['a', 'b']:
a->ab,b->a
```

Automorphisms can be composed, inverted (note that there is not test of invertibility upon creation), exponentiated, applied to free group elements.

```
sage: phi=FreeGroupAutomorphism('a->ab,b->ac,c->a')
sage: phi=FreeGroupAutomorphism('a->c,b->ba,c->bcc')
sage: print phi*psi
a->a,b->acab,c->acaa
sage: print phi.inverse()
a->c,b->Ca,c->Cb
sage: print phi**3
a->abacaba,b->abacab,c->abac
sage: phi('aBc')
word: abC
```

Note that there is a list of pre-defined automorphisms of free groups taken from the litterature:

```
sage: print free_group_automorphisms.Handel_Mosher_inverse_with_same_lambda()
a->b,b->c,c->Ba
```

Also Free group automorphisms can be obtained as composition of elementary Nielsen automorphisms (of the form $a \mapsto ab$). Up to now they are rather called Dehn twists.

```
sage: F=FreeGroup(3)
sage: print F.dehnt_twist('a','c')
a->ac, b->b, c->c
sage: print F.dehn_twist('A','c')
a->Ca,b->b,c->c
sage: print F.dehn_twist('a','b',on_left=True)
a->ba,b->b,c->c
```

If the free group as even rank $N = 2g$, then it is the fundamental group of an oriented surface ofgenus g with one boundary component. In this case the mapping class group of $S_{g,1}$ is a subgroup of the outer automorphism group of F_N and it is generated by a collection of $3g-1$ Dehn twists along curves. Those Dehn twistss are accessed through:

```
sage: F=FreeGroup(4)
sage: print F.surface_dehn_twist(2)
a->a,b->ab,c->acA,d->adA
```

Similarly the group B_N is a subgroup of $\mathrm{Aut}(F_N)$ and its usual generators are obtained by:

```
sage: F=FreeGroup(3)
sage: print F.braid_automorphism(0)
a->c,b->b,c->caC
```

Finally for statistical purpose, one can access random automorphisms or random mapping classes or random braids. The random elements are obtained by composition of a given number of randomly chosen generators of these groups.

```
sage: F=FreeGroup(4)
sage: F.random_automorphism(8)
```

4 Graphs and maps

Graphs and maps are used to represent free group automorphisms. A graph here is a GraphWithInverses: it has a set of vertices and a set of edges in one-to-one correspondance with the letters of an AlphabetWithInverses: each non-oriented edge is a pair $\{e, \bar{e}\}$ of a letter of the alphabet and its inverse. This is complient with Serre's view. As the alphabet has a set of positive letters there is a default choice of orientation for edges.

The easiest graph is the :

```
sage: A=AlphabetWithInverses(3)
sage: G=GraphWithInverses.rose_graph(A)
sage: print G
Graph with inverses: a: 0->0, b: 0->0, c: 0->0
sage: G.plot()
```

Otherwise a graph can be given by a variety of inputs like a list of edges, etc. Graphs can easily be plotted. Note that `plot()` tries to lower the number of accidental crossing of edges, using some thermodynamics and randomness, thus two calls of `plot()` may output two different figures.

A number of operations on graphs are defined: subdividing, folding, collapsing edges, etc.

Graphs come with maps between them: a map is a continuous map from a graph to another which maps vertices to vertices and edges to edge-paths. Again they can be given by a variety of means. As Graph maps are intended to represent free group automorphisms a simple way to create a graph map is from a free group automorphism:

```
sage: phi=free_group_automorphisms.tribonacci()
sage: print phi.rose_representative()
Topological representative:
Marked graph: a: 0->0, b: 0->0, c: 0->0
Marking: a->a, b->b, c->c
Edge map: a->ab, b->ac, c->a
```

Remark that by default the rose graph is **marked**: it comes with a marking from the rose (itself, but you should think of that one as fixed) to the graph. Here the graph map is a graph self map as the source and the target are the same.

Graph maps can also be folded, subdivided, etc. If the graphs are marked then those operations will carry on the marking.

5 Train-tracks

The main feature and the main achievement of the program is to compute train-track representative for (outer) automorphisms of free groups. `phi.train_track()` computes a train-track representative for the (outer) automorphism phi. This train-track can be either an absolute train-track or a relative train-track. The celebrated theorem of Bestvina and Feighn [**?**] assures that if phi is fully irreducible (iwip) then there exists an absolute train-track representing phi.

The `train-track(relative=False)` method will terminate with either an absolute train-track or with a topological representative with a reduction: an invariant strict subgraph with non-trivial fundamental group.

One more feature of train-tracks (absolute or relative) is to lower the number of Nielsn paths. Setting the `stable=True` option will return a train-track with at most one indivisible Nielsen path (per exponential stratum if it is a relative train-track).

6 More on free group automorphisms

Using the `train-track()` method our program can decide wether an automorphism is fully irreducible or not. If it is iwip, one can compute the **index**, index-list or **ideal Whitehead graphs**. Not that these computations are done using an absolute expanding train-track representative: they can be use for a broader class than just iwip automorphisms.

7 Convex cores, curve complex and more

Index