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

Thierry Coulbois

March 17, 2016

1 Introduction

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

It is primarily intended to implement the computation of a train-track representative for automorphisms of free groups as introduced by M. Bestvina and M. Handel [1].

Sage is based on Python. This is an object oriented language: the **post-fix 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 distribution of Sage. We recommand that you run the latest version. Sage is constantly under development, so check regularly at `https://sagemath.org`. Then you need to download the files from `https://github.com/coulbois/sage-train-track`. We hope that soon this package will be part of the Sage distribution.

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: w.reduced()
   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: w.reduced()
KeyError: 'x'
sage: w=F(['x0','X0','x1']); w
word: x0,X0,x1
sage: w.reduced()
word: x1
```

3.3 Free group automorphisms

The creation (and 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 no 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
```

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: FreeGroupAutomorphism.dehn_twist(F,'a','c')
a->ac, b->b, c->c
sage: FreeGroupAutomorphism.dehn_twist(F,'A','c')
Automorphism of the Free group over ['a', 'b', 'c']:
a->Ca,b->b,c->c
sage: print FreeGroupAutomorphism.dehn_twist(F,'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 of genus 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 twists are accessed through:

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

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

```
sage: F=FreeGroup(3)
sage: print FreeGroupAutomorphism.braid_automorphism(F,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 [5, 6]. As the alphabet has a set of positive letters there is a default choice of orientation for edges.

The easiest graph is the rose:

```
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. But, as of now, not all Stallings [7] moves are implemented.

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()
GraphSelfMap:
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.

Note that to associate an automorphism to a graph self map that is a homotopy equivalence we need to fix a base point to compute the fundamental group. Thus if we do not fix the base point we only get an outer automorphism of the free group. However, the program do not handle directly outer automorphism, rather `f.automophism()` returns an automorphism but with no guarantee on how the base is chosen, thus this automorphism is an arbitrary representative of the graph self map f. Moreover, if the base graph is not marked, then the automorphism is only defined up to conjugacy in $\mathrm{Out}(F_N)$. In this case `f.automorphism()` returns an arbitrary automorphism in the conjugacy class. We provide a `phi.simple_outer_representative()` which return an automorphism in the outer class of ϕ with the smallest possible length of images.

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 M. Bestvina and M. Handel [1] assures that if Φ is fully irreducible (iwip) then there exists an absolute train-track representing Φ.

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 Nielsen 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).

5.1 Examples

Let's start with building absolute train-tracks.

```
sage: phi=free_group_automorphisms.tribonacci()
sage: phi.train_track()
    Train-track map:
    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
    Irreducible representative
```

Indeed Tribonacci automorphism $\phi : a \mapsto ab,\ b \mapsto ac,\ c \mapsto a$ is a positive automorphism (also called substitution), and thus it defines a map from the rose to itself which is a train-track map. Note that here the output is a `TrainTrackMap`.

```
sage: phi=FreeGroupAutomorphism("a->ab,b->ac,c->c")
sage: phi.train_track(relative=False)
    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->c
    Strata: [set(['c']), set(['a', 'b'])]
```

Here the automorphism is not irreducible (it fixes the free group element c). And the algorithm correctly detect that by returning a stratified graph map. Although the rose representative is reducible, it is a train track map (because the automorphism is positive). But this is not detected by the `train_track()` method. We provide a `is_train_track()` method to test that.

```
sage: phi=FreeGroupAutomorphism("a->ab,b->ac,c->c")
sage: f=phi.rose_representative()
sage: f.is_train_track()
True
```

You can promote this map to become a `TrainTrackMap` by using `TrainTrackMap(f)`. This can be useful to compute Nielsen paths of such reducible train-track maps (but this may cause infinite loops in the program).

Reducible automorphisms always have a relative train-track representative.

```
sage: phi=FreeGroupAutomorphism("a->ab,b->ac,c->c")
sage: phi.train_track()
Graph self map:
Marked graph: a: 2->0, b: 1->0, c: 0->0, d: 1->0, e: 2->0
Marking: a->Ea, b->Db, c->c
Edge map: a->b, b->ac, c->c, d->e, e->dAe
Strata: [set(['c']), set(['a', 'b']), set(['e', 'd'])]
```

(compare with the above example and note that the default option is `relative=True`). Ask for details of the computation by setting option `verbose=1` (or 2, or more).

The default option for this `train_track` method is to set `stable=True`, meaning that it looks for a stable train-track.

5.2 Train-tracks and graph maps

In the previous section we computed train-track representatives for automorphisms of free group. The process goes by building a graph self map on the rose to represent the automorphism (this is called a topological representative and then perform operations on this graph self map.

The graph on which the topological representative is built can be any kind of our graphs: `GraphWithInverses`, `MarkedGraph`, `MetricGraph`, `MarkedMetricGraph`. If the graph is not marked, then one give up the possibility to recover the original outer automorphism from the train-track. Indeed, all outer automorphisms in a conjugacy class in $\mathrm{Out}(F_N)$ can be represented as the same homotopy equivalence on a graph.

The train-track algorithm can be called directly on a graph self map `f.train_track()` with the same options as for automorpism but f will not be promoted to become a `TrainTrackMap` even if it could. One can access intermediate operations like `f.stabilize()`, `f.reduce()`, etc.

5.3 Nielsen paths

Nielsen paths are a main tool to refine the understanding of train-tracks and of automorphisms of free groups. A Nielsen path for a graph self map f is a path homotopic to its image relative to its endpoints. In our context, we only compute and use Nielsen paths in the case of train-track maps (or relative train-track maps).

Nielsen paths of a graph self map `f` can be computed `f.indivisible_nielsen_paths()`. The output is a list of pairs `(u,v)` of paths in the domain of `f`. The paths `u` and `v` starts at the same vertex and the ends of the Nielsen path are inside the last edges of `u` and `v`. We also provide the computation of periodic Nielsen paths, that-is-to-say Nielsen paths of iterates of `f`. In this case a Nielsen path is coded by `((u,v),period)`. To build longer Nielsen paths we need to concatenate the indivisible ones and for that we need to encode the endpoints of periodic Nielsen paths. This normal form for points inside edges is a little tricky and can be obtained using `TrainTrackMap.periodic_point_normal_form()`.

6 More on free group automorphisms

We implemented the computation of other invariants for iwip automorphisms of free groups. Beware, that Python and Sage let you check the requirements: computing the index of a reducible automorphism may cause errors or infinite loops by the program.

A graph self map as Whitehead graphs at each vertex and thanks to Brian Mann, they can be computed. The Whitehead graph of a graph self map $f : G \to G$ at a vertex v as the set of germs of edges outgoing from v as vertices and as an edge for a germ from another if and only this turn is taken by the iterated image of an edge. Stable Whitehead graphs are also available: they only keep germs of edges which are periodic.

Finally the ideal Whitehead graph is an invariant of iwip automorphisms. And we can compute them. From the ideal Whitehead graph one can compute the index list and the index of an iwip automorphism of a free group.

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

7.1 Metric simplicial trees and Outer space

The programm is also designed to handle trees in Outer space as well as simplicial trees in the boundary of Outer space.

Recall that M. Culler and K. Vogtmann [2] introduced the Outer space of a free group F_N which we denote by CV_N. Outer space is made of simplicial metric trees T with a free minimal action of the free group F_N by isometries. Alternatively a point in Outer space is a marked metric graph T/F_N.

Our classes `MetricGraph` and `MarkedMetricGraph` allow us to handle points in Outer space. In a metric graph edges of length 0 can be used as an artefact to code simplicial trees with non-free action. For instance

```
sage: A = AlphabetWithInverses(3)
sage: G = MarkedMetricGraph.splitting(2,A)
```

```
sage: print G
Marked graph: a: 0->0, b: 0->0, c: 1->1, d: 0->1
Marking: a->a, b->b, c->dcD
Length: a:0, b:0, c:0, d:1
```

This graphs codes the splitting of the free group $F_3 = F_2 * \mathbb{Z}$. HNN-splittings are also available: `MarkedMetricGraph.HNN_splitting()`. Thus the metric graphs (with edges of length 0) are a convenient tool to work with the splitting complex.

Let us emphasize that the splitting complex of a free group is becoming a popular tool after being proved hyperbolic by M. Handel and L. Mosher [4].

In the geometric situation, these non-free trees can be used to encode arcs in the arc complex for a surface $S_{g,1}$ of genus g with one puncture, ideal arcs (a curve from the puncture to itself) are in one-to-one correspondance with splittings of the free group $\pi_1(S_{g,1})$. They can also be used to study curve diagram in the context of braid groups.

7.2 Convex cores

We also implemented the computation of V. Guirardel [3] convex core of two simplicial trees in outer space and its boundary. The convex core is a square complex inside the cartesian product $T_0 \times T_1$ of two trees with action of the free group. Here it is encoded by its quotient $C(T_0 \times T_1)/F_N$ which is a finite square complex. We have the convention that the convex core is connected and thus we give up unicity: instead we include twice light squares inside the core.

The first way to create a convex core is by using a free group automorphism `phi`. Then `ConvexCore(phi)` returns the convex core of the Cayley graph T_0 of the free group with the tree T_1 which is the same as T_0 but with the action twisted by `phi`.

```
sage: phi = FreeGroupAutomorphism.tribonacci()**3
sage: C = ConvexCore(phi)
sage: C.squares()
[[5, 0, 2, 1, 'b', 'a'],
 [5, 6, 4, 1, 'c', 'a'],
 [7, 0, 2, 3, 'c', 'a'],
 [3, 2, 6, 5, 'c', 'b']]
sage: C.one_squeleton(side=1)
Looped multi-digraph on 8 vertices
```

The second way involves creating the two trees. This requires the creation of two marked graphs, which can be a little teddious, but some methods shorten the typesetting.

```
sage: A = AlphabetWithInverses(3)
sage: G0 = MarkedGraph.rose_marked_graph(A)
sage: G1 = MarkedGraph(GraphWithInverses.valence_3(3))
sage: G1.precompose(phi)
sage: C = ConvexCore(G0,G1)
sage: C.volume()
```

Remark that if the automorphism is a mapping class and the trees are transverse to ideal curves then the convex core (as a CW-complex) is homeomorphic to the surface.

Index

References

[1] Mladen Bestvina and Michael Handel, *Train tracks and automorphisms of free groups*, Ann. of Math. (2) **135** (1992), no. 1, 1–51. MR MR1147956 (92m:20017)

[2] Marc Culler and Karen Vogtmann, *Moduli of graphs and automorphisms of free groups.*, Invent. Math. **84** (1986), 91–119 (English).

[3] Vincent Guirardel, *Cœur et nombre d'intersection pour les actions de groupes sur les arbres*, Ann. Sci. École Norm. Sup. (4) **38** (2005), no. 6, 847–888. MR MR2216833 (2007e:20055)

[4] Michael Handel and Lee Mosher, *The free splitting complex of a free group, I: hyperbolicity*, Geom. Topol. **17** (2013), no. 3, 1581–1672. MR 3073931

[5] Jean-Pierre Serre, *Arbres, amalgames,* SL_2, Société Mathématique de France, Paris, 1977, Avec un sommaire anglais, Rédigé avec la collaboration de Hyman Bass, Astérisque, No. 46. MR 0476875

[6] ______, *Trees*, Springer Monographs in Mathematics, Springer-Verlag, Berlin, 2003, Translated from the French original by John Stillwell, Corrected 2nd printing of the 1980 English translation. MR 1954121

[7] John R. Stallings, *Topology of finite graphs*, Invent. Math. **71** (1983), no. 3, 551–565. MR MR695906 (85m:05037a)