`braidlab` user's guide

Jean-Luc Thiffeault

Contents

1 Installing `braidlab` **2**
- 1.1 Precompiled packages . . . 2
- 1.2 Cloning the repository . . . 2
- 1.3 Setting Matlab's path . . . 2
- 1.4 Testing your installation . . . 3

2 A tour of `braidlab` **3**
- 2.1 The `braid` class . . . 3
  - 2.1.1 Constructor and elementary operations . . . 3
  - 2.1.2 Topological entropy and complexity . . . 5
  - 2.1.3 Representation and invariants . . . 7
- 2.2 Constructing a braid from data . . . 8
- 2.3 The `loop` class . . . 11
  - 2.3.1 Loop coordinates . . . 11
  - 2.3.2 Acting on loops with braids . . . 13
- 2.4 Loop coordinates for a braid . . . 15

3 Side note: On filling-in punctures **16**

Acknowledgments **19**

References **21**

Index **22**

1 Installing `braidlab`

1.1 Precompiled packages

Some zip and tar files of the precompiled latest released version are available at `http://bitbucket.org/jeanluc/braidlab/downloads`. If one of those suits your system, then download and untar/unzip.

1.2 Cloning the repository

If you prefer to have the latest (possibly unstable) development version, and know how to compile Matlab MEX files on your system, then you can clone the Mercurial source repository with the terminal command

```
# hg clone https://bitbucket.org/jeanluc/braidlab
```

assuming Mercurial is installed on your system. After this finishes, type

```
# cd braidlab; make all
```

to compile the MEX files. Note that you can still use `braidlab` even if you're unable to compile the MEX files, but some commands will be unavailable or run more slowly.

1.3 Setting Matlab's path

The package `braidlab` is defined inside a Matlab namespace, which are specified as subfolders beginning with a '`+`' character. The Matlab path must contain the folder that contains the subfolder `+braidlab`, and not the `+braidlab` folder itself:

```
>> addpath 'path to folder containing +braidlab'
```

To execute a braidlab *function*, either call it using the syntax `braidlab.`*function*, or import the whole namespace:

```
>> import braidlab.*
```

This allows invoking *function* by itself, without the `braidlab` prefix. For the remainder of this document, we assume this has been done and omit the `braidlab` prefix. The `addpath` and `import` commands can be added to `startup.m` to ensure they are executed at the start of every Matlab session.

1.4 Testing your installation

To check that everything is working, `braidlab` includes a testsuite. From Matlab, change to the `testsuite` folder, and run

```
>> test_braidlab
```

making sure the path is set properly (Section 1.3). Note that running the testsuite requires Matlab version 2013a or later.

2 A tour of `braidlab`

2.1 The `braid` class

2.1.1 Constructor and elementary operations

`braidlab` defines a number of classes, most importantly `braid` and `loop`. The braid $\sigma_1\sigma_2^{-1}$ is constructed with

```
>> a = braid([1 -2])   % defaults to 3 strings

a = < 1 -2 >
```

which defaults to the minimum required strings, 3. The same braid on 4 strings is constructed with

```
> a4 = braid([1 -2],4)   % force 4 strings

a4 = < 1 -2 >
```

Two braids can be multiplied:

```
>> a = braid([1 -2]); b = braid([1 2]);
>> a*b, b*a

ans = < 1 -2  1  2 >

ans = < 1  2  1 -2 >
```

Powers can also be taken, including the inverse:

```
>> a^5, inv(a), a*a^-1

ans = < 1 -2  1 -2  1 -2  1 -2  1 -2 >
```

```
ans = < 2 -1 >

ans = < 1 -2  2 -1 >
```

Note that this last expression is the identity braid, but is not simplified. The method `compact` attempts to simplify the braid:

```
>> compact(a*a^-1)

ans = < e >
```

The method `compact` is based on the heuristic algorithm of Bangert *et al.* (2002), since finding the braid of minimum length in the standard generators is in general difficult (Paterson & Razborov, 1991). Hence, there is no guarantee that in general `compact` will find the identity braid, even though it do so here. To really test if a braid is the identity (trivial braid), use the method `istrivial`:

```
>> istrivial(a*a^-1)

ans = 1
```

The number of strings is

```
>> a.n

ans = 3
```

Note that

```
>> help braid
```

describes the class `braid`. To get more information on the `braid` constructor, invoke

```
>> help braid.braid
```

which refers to the method `braid` within the class `braid`. (Use `methods(braid)` to list all the methods in the class.) There are other ways to construct a `braid`, such as using random generators, here a braid with 5 strings and 10 random generators:

```
>> braid('random',5,10)

ans = < 1  4 -4  2  4 -1 -2  4  4  4 >
```

The constructor can also build some standard braids:

```
>> braid('halftwist',5)

ans = < 4  3  2  1  4  3  2  4  3  4 >
```

In Section 2.2 we will also show how to construct a braid from a trajectory data set.

The `braid` class also handles equality of braids:

```
>> a = braid([1 -2]); b = braid([1 -2 2 1 2 -1 -2 -1]);
>> a == b

ans = 1
```

These are the same braid, even though they appear different from their generator sequence (Birman, 1975). Equality is determined efficiently by acting on loop coordinates (Dynnikov, 2002), as described by Dehornoy (2008). See Sections 2.3–2.4 for more details. If for some reason lexicographic (generator-per-generator) equality of braids is needed, use the method `lexeq(b1,b2)`.

We can extract a subbraid by choosing specific strings: for example, if we take the 4-string braid $\sigma_1\sigma_2\sigma_3^{-1}$ and discard the third string, we obtain $\sigma_1\sigma_2^{-1}$:

```
>> a = braid([1 2 -3]);
>> subbraid(a,[1 2 4])   % subbraid using strings 1,2,4

ans = < 1 -2 >
```

The opposite of subbraid is the *tensor product*, the larger braid obtained by laying two braids side-by-side (Kassel & Turaev, 2008):

```
>> a = braid([1 2 -3]); b = braid([1 -2]);
>> tensor(a,b)

ans = < 1  2 -3  5 -6 >
```

Here, the tensor product of a 4-braid and a 3-braid has 7 strings. The generators $\sigma_1\sigma_2^{-1}$ of `b` became $\sigma_5\sigma_6^{-1}$ after re-indexing so they appear to the right of `a`.

2.1.2 Topological entropy and complexity

There are a few methods that exploit the connection between braids and homeomorphisms of the punctured disk. Braids label *isotopy classes* of homeomorphisms, so we can assign a topological entropy to a braid:

```
>> entropy(braid([1 2 -3]))

ans = 0.8314
```

The entropy is computed by iterated action on a loop (Moussafir, 2006). This can fail if the braid is finite-order or has very low entropy:

```
>> entropy(braid([1 2]))
Warning: Failed to converge to requested tolerance; braid is
   likely finite-order or has low entropy.
> In braid.entropy at 89

ans = 0
```

To force the entropy to be computed using the Bestvina–Handel train track algorithm Bestvina & Handel (1995), we add an optional parameter:

```
>> entropy(braid([1 2]),'trains')

ans = 0
```

Note that for large braids the Bestvina–Handel algorithm is impractical. But when applicable it can also determine the Thurston–Nielsen type of the braid (Fathi *et al.*, 1979; Thurston, 1988; Casson & Bleiler, 1988; Boyland, 1994):

```
>> tntype(braid([1 2 -3]))

ans = pseudo-Anosov
>> tntype(braid([1 2]))

ans = finite-order
>> tntype(braid([1 2],4))  % reducing curve around 1,2,3

ans = reducible
```

`braidlab` uses Toby Hall's implementation of the Bestvina–Handel algorithm (Hall, 2012).

The topological entropy is a measure of braid complexity that relies on iterating the braid. It gives the maximum growth rate of a 'rubber band' anchored on the braid, as the rubber band slides up many repeated copies of the braid. For finite-order braids, this will converge to zero. The *geometric complexity* of a braid (Dynnikov & Wiest, 2007), is defined in terms of the $\log_2$ of the number of intersections of a set of curves with the real axis, after one application of the braid:

```
>> complexity(braid([1 -2]))

ans = 2
>> complexity(braid([1 2]))

ans = 1.5850
```

See Section 2.3 or '`help braid.complexity`' for details on how the geometric complexity is computed.

2.1.3 Representation and invariants

There are a few remaining methods in the braid class, which we describe briefly. The reduced Burau matrix representation (Burau, 1936; Birman, 1975) of a braid is obtained with the method `burau`:

```
>> burau(braid([1 -2]),-1)

ans = 1     -1
     -1      2
```

where the last argument (-1) is the value of the parameter t in the Laurent polynomials that appear in the entries of the Burau matrices. With access to Matlab's wavelet toolbox, we can use actual Laurent polynomials as the entries:

```
>> B = burau(braid([1 -2]),laurpoly(1,1))

     | - z^(+1)          z^(+1)      |
     |                               |
 B = |                               |
     |                               |
     |   - 1         + 1 - z^(-1)    |
```

but the matrix is now given as a cell array, each entry containing a `laurpoly` object:

```
>> B{2,2}

ans(z) = + 1 - z^(-1)
```

The reduced Burau matrix of a braid can be used to compute the *Alexander–Conway polynomial* (or Alexander polynomial for short) of its closure (see Section 2.2 for more on the closure of a braid). For instance, the trefoil knot is given by the closure of the braid σ_1^3 (Weisstein, 2013), which gives a Laurent polynomial

```
>> alexpoly(braid([1 1 1]))

ans(z) = + z^(+1) - 1 + z^(-1)
```

The figure-eight know is the closure of $(\sigma_1\sigma_2^{-1})^2$:

```
>> alexpoly(braid([1 -2 1 -2]))

ans(z) = - z^(+1) + 3 - z^(-1)
```

The Alexander polynomial is a knot invariant, so it can be used to determine when two knots are not the same.

The method `perm` gives the permutation of strings corresponding to a braid:

```
>> perm(braid([1 2 -3]))

ans = 2  3  4  1
```

If the strings are unpermuted, then the braid is *pure*, which can also be tested with the method `ispure`.

Finally, the *writhe* of a braid is the sum of the powers of its generators. The writhe of $\sigma_1^{+1}\sigma_2^{+1}\sigma_3^{-1}$ is $+1+1-1=1$:

```
>> writhe(braid([1 2 -3]))

ans = 1
```

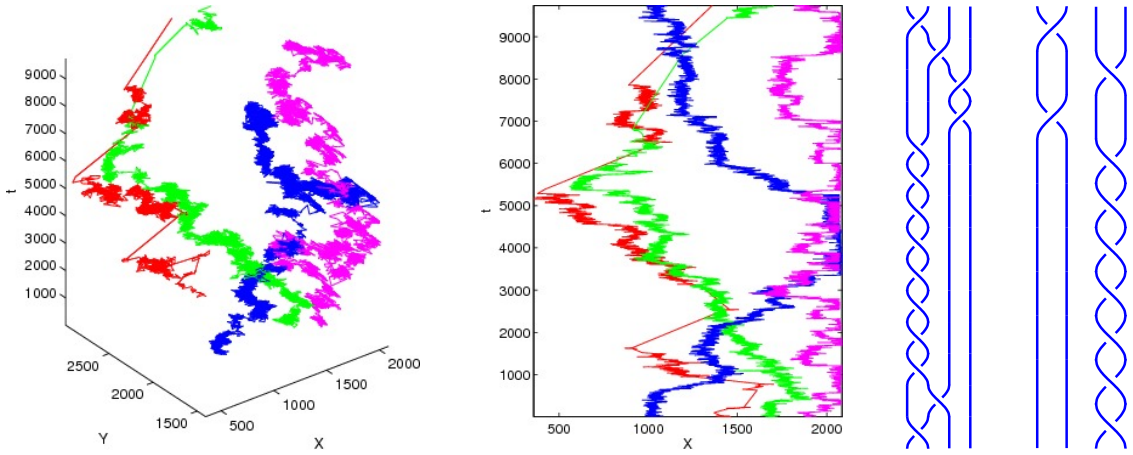
The writhe is a braid invariant.

2.2 Constructing a braid from data

One of the main purposes of `braidlab` is to analyze two-dimensional trajectory data using braids. We can assign a braid to trajectory data by looking for *crossings* along a projection line (Thiffeault, 2005, 2010). The `braid` constructor allows us to do this easily.

The folder `testsuite` contains a dataset of trajectories, from laboratory data for granular media (Puckett *et al.*, 2012). From the `testsuite` folder, we load the data:

```
>> clear; load testdata
>> whos
  Name          Size                Bytes  Class     Attributes
```



(a) (b) (c) (d)

Figure 1: (a) A dataset of four trajectories, (b) projected along the X axis. (c) The compacted braid $\sigma_1^{-1}\sigma_2^{-1}\sigma_1^{-8}\sigma_3^2\sigma_2\sigma_1$ corresponding to the X projection in (b). (d) The compacted braid $\sigma_3^{-7}\sigma_1\sigma_3^{-1}\sigma_1$ corresponding to the Y projection, with closure enforced. The braids in (c) and (d) are conjugate.

```
  XY          9740x2x4                623360   double
  ti             1x9740                77920   double
```

Here `ti` is the vector of times, and `XY` is a three-dimensional array: its first component specifies the timestep, its second specifies the X or Y coordinate, and its third specifies one of the 4 particles. Figure 1(a) shows the X and Y coordinates of these four trajectories, with time plotted vertically. Figure 1(b) shows the same data, but projected along the X direction. To construct a braid from this data, we simply execute

```
>> b = braid(XY);
>> b.length

ans = 894
```

This is a very long braid! But Figure 1(b) suggests that this is misleading: many of the crossings are 'wiggles' that cancel each other out. Indeed, if we attempt to shorten the braid:

```
>> b = compact(b)
```

```
b = < -1 -2 -1 -1 -1 -1 -1 -1 -1 -1  3  3  2  1 >
>> b.length

ans = 14
```

we find the number of generators (the length) has dropped to 14! We can then plot this shortened braid as a braid diagram using `plot(b)` to produce Figure 1(c). The braid diagram allows us to see topological information clearly, such as the fact that the second and third particles undergo a large number of twists around each other; we can check this by creating a subbraid with only those two strings:

```
>> subbraid(bX,[2 3])

ans = < -1 -1 -1 -1 -1 -1 -1 -1 >
```

which shows that the winding number between these two strings is -4.

The braid was constructed from the data by assuming a projection along the X axis (the default). We can choose a different projection by specifying an optional angle for the projection line; for instance, to project along the Y axis we invoke

```
>> b = braid(XY,pi/2);   % project onto Y axis
>> b.length

ans = 673
>> b.compact

ans = < -3 -3 -3 -3 -3 -3 -3  1 -3 >
```

In general, a change of projection line only changes the braid by conjugation (Boyland, 1994; Thiffeault, 2010). We can test for conjugacy:

```
>> bX = compact(braid(XY,0)); bY = compact(braid(XY,pi/2));
>> conjtest(bX,bY)   % test for conjugacy of braids

ans = 0
```

The braids are not conjugate. This is because our trajectories do not form a 'true' braid: the final points do not correspond exactly with the initial points, as a set. If we truly want a rotationally-conjugate braid out of our data, we need to enforce a closure method:

```
>> XY = closure(XY);   % close braid and avoid new crossings
```

```
>> bX = compact(braid(XY,0)), bY = compact(braid(XY,pi/2))

bX = < -1 -2 -1 -1 -1 -1 -1 -1 -1 -1  3  3  2  1 >

bY = < -3 -3 -3 -3 -3 -3 -3  1 -3  1 >
```

This default closure simply draws line segments from the final points to the initial points in such a way that no new crossings are created in the X projection. Hence, the X-projected braid `bX` is unchanged by the closure, but here the Y-projected braid `bY` is longer by one generator (`bY` is plotted in Figure 1(d)). This is enough to make the braids conjugate:

```
>> [~,c] = conjtest(bX,bY)  % ~ means discard first return arg

c = < 3  2 >
```

where the optional second argument `c` is the conjugating braid, as we can verify:

```
>> bX == c*bY*c^-1

ans = 1
```

There are other ways to enforce closure of a braid (see `help closure`), in particular `closure(XY,'mindist')`, which minimizes the total distance between the initial and final points.

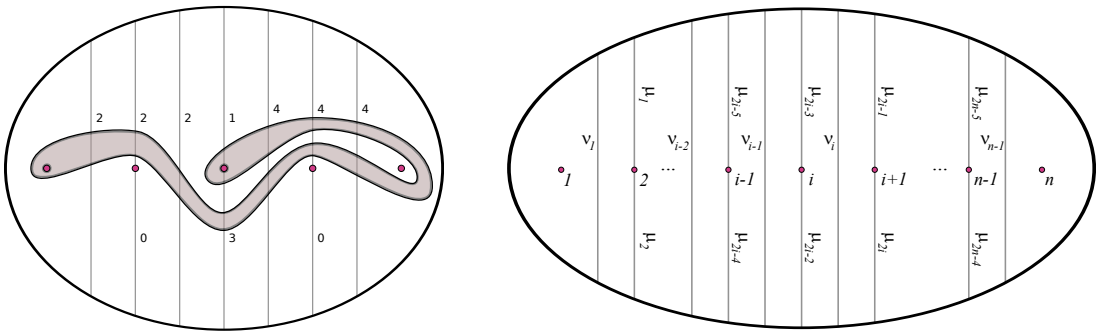
Note that `conjtest` uses the library *CBraid* (Cha, 2011) to first convert the braids to Garside canonical form (Birman & Brendle, 2005), then to determine conjugacy. This is very inefficient, so is impractical for large braids.

2.3 The `loop` class

2.3.1 Loop coordinates

A simple closed loop on a disk with 5 punctures is shown in Figure 2(a). We consider equivalence classes of such loops under homotopies relative to the punctures. In particular, the loops are *essential*, meaning that they are not null-homotopic or homotopic to the boundary or a puncture. The *intersection numbers* are also shown in Figure 2(a): these count the minimum number of intersections of an equivalence class of loops with the fixed vertical lines shown. For n punctures, we define the intersection numbers μ_i and ν_i in Figure 2(b).

Any given loop will lead to a unique set of intersection numbers, but a general collection of intersection numbers do not typically correspond to a loop. It is therefore



(a) (b)

Figure 2: (a) A simple close loop in a disk with $n = 5$ punctures. (b) Definition of intersection numbers μ_i and ν_i. [From Thiffeault (2010).]

more convenient to define

$$a_i = \tfrac{1}{2}\left(\mu_{2i} - \mu_{2i-1}\right), \qquad b_i = \tfrac{1}{2}\left(\nu_i - \nu_{i+1}\right), \qquad i = 1, \ldots, n-2. \tag{1}$$

We then combine these in a vector of length $(2n-4)$,

$$\boldsymbol{u} = (a_1, \ldots, a_{n-2}, b_1, \ldots, b_{n-2}), \tag{2}$$

which gives the *loop coordinates* (or *Dynnikov coordinates*) for the loop. (Some authors such as Dehornoy (2008) give the coordinates as $(a_1, b_1, \ldots, a_{n-2}, b_{n-2})$.) There is now a bijection between $\mathbb{Z}^{2n-4}$ and essential simple closed loops (Dynnikov, 2002; Moussafir, 2006; Hall & Yurttaş, 2009; Thiffeault, 2010). Actually, *multiloops*: loop coordinates can describe unions of disjoint loops (see Section 2.4).[1]

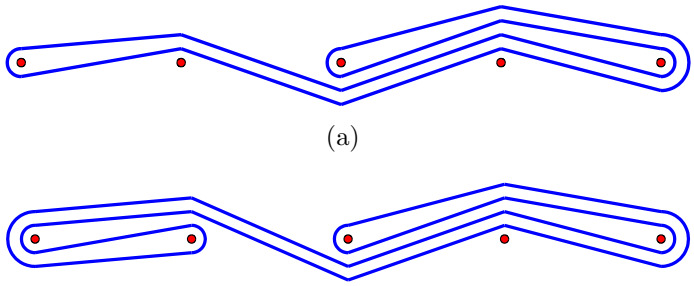
Let's create the loop in Figure 2(a) as a `loop` object:

```
>> l = loop([-1 1 -2 0 -1 0])

l = (( -1 1 -2 0 -1 0 ))
```

Figure 3(a) shows the output of the `plot(l)` command. We can convert from loop coordinates to intersection numbers with

```
>> intersec(l)

ans = 2 0 1 3 4 0 2 2 4 4    % [mu1 ... mu6 nu1 ... nu4]
```

[1]Here we use multiloop as a convenient mnemonic. The technical term is *integral lamination*: a set of disjoint non-homotopic simple closed curves (Moussafir, 2006).



(a)

(b)

Figure 3: (a) The loop `((-1 1 -2 0 -1 0))`. (b) The braid generator σ_1^{-1} applied to the loop in (a).

which returns $\mu_1 \dots \mu_{2n-4}$ followed by $\nu_1 \dots \mu_{n-1}$, as defined in Figure 2(b).

We can also extract the loop coordinates from a `loop` object using the methods `a`, `b`, and `ab`:

```
>> l = loop ([ -1 1 -2 0 -1 0]);
>> l.a

ans = -1      1     -2
>> l.b

ans =  0     -1      0
>> [a,b] = l.ab

a = -1      1     -2
b =  0     -1      0
```

As for braids, `l.n` returns the number of punctures (or strings).

2.3.2 Acting on loops with braids

Now we can act on this loop with braids. For example, we define the braid `b` to be σ_1^{-1} with 5 strings, corresponding to the 5 punctures, and then act on the loop `l` by using the multiplication operator:

```
>> b = braid([-1],5);    % one generator with 5 strings
>> b*l                   % act on a loop with a braid
```

```
ans = (( -1  1 -2  1 -1  0 ))
```

Figure 3(b) shows `plot(b*l)`. The first and second punctures were interchanged counterclockwise (the action of σ_1^{-1}), dragging the loop along.

The minimum length of an equivalence class of loops is determined by assuming the punctures are one unit of length apart and have zero size. After pulling tight the loop on the punctures, it is then made up of unit-length segments. The minimum length is thus an integer. For the loop in Figure 3(a),

```
>> minlength(l)

ans = 12
```

Another useful measure of a loop's complexity is its minimum intersection number with the real axis (Moussafir, 2006; Hall & Yurttaş, 2009; Thiffeault, 2010), which for this loop is the same as its minimum length:

```
>> intaxis(l)

ans = 12
```

The `intaxis` method is used to measure a braid's geometric complexity, as defined by Dynnikov & Wiest (2007).

Sometimes we wish to study a large set of different loops. The loop constructor vectorizes:

```
>> ll = loop([-1 1 -2 0; 1 -2 3 4])

ll = (( -1  1 -2  0 ))
     (( 1 -2  3  4 ))
```

We can then, for instance, compute the length of every loop:

```
>> minlength(ll)

ans = 14
      34
```

or even act on all the loops with the same braid:

```
>> b = braid([1 -2);
>> b*ll

ans = (( 2  1 -2  1 ))
      (( 5  -2  -3  11 ))
```

Some commands, such as `plot`, do not vectorize. Different loops can then be accessed by indexing, such as `plot(ll(2))`.

The `entropy` method of the `braid` class (Section 2.1) computes the topological entropy of a braid by repeatedly acting on a loop, and monitoring the growth rate of the loop. For example, let us compare the entropy obtained by acting 100 times on an initial loop, compared with the `entropy` method:

```
>> b = braid([1 2 3 -4]);
% apply braid 100 times to l, then compute growth of length
>> log(minlength(b^100*l)/minlength(l)) / 100

ans = 0.7637
>> entropy(b)

ans = 0.7672
```

The entropy value returned by `entropy(b)` is more precise, since that method monitors convergence and adjusts the number of iterations accordingly.

2.4 Loop coordinates for a braid

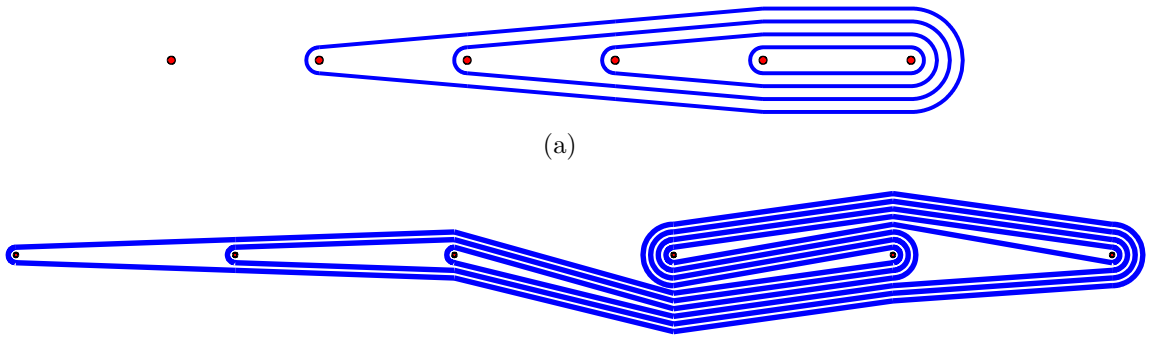
The command `loop(n)` returns a *canonical set of loops* for n punctures:

```
>> loop(5)

ans = (( 0  0  0  0 -1 -1 -1 -1 ))
```

This multiloop is depicted in Figure 4(a). Note that the multiloop returned by `loop(5)` actually has 6 punctures! The rightmost puncture is meant to represent the boundary of a disk, or a base point for the fundamental group on a sphere with n punctures. The loops form a generating set for the fundamental group of the disk with n punctures.

The canonical set of loops allows us to define loop coordinates for a braid, which is a unique normal form. The canonical loop coordinates for braids exploit the fact that two braids are equal if and only if they act the same way on the fundamental group of the disk (Dehornoy, 2008). Hence, if we take a braid and act on `loop(5)`,

```
>> b = braid([1 2 3 -4]);
>> b*loop(5)

ans = (( 0  0  3 -1 -1 -1 -4  3 ))
```



(a)

(b)

Figure 4: (a) The multiloop created by `loop(5)`. (b) The multiloop `b*loop(5)`, where `b` is the braid $\sigma_1\sigma_2\sigma_3\sigma_4^{-1}$.

then the set of numbers `(( 0 0 3 -1 -1 -1 -4 3 ))` can be thought of as *uniquely* characterizing the braid. It is this property that is used to rapidly determine equality of braids. (The loop `b*loop(5)` is plotted in Figure 4(b).) The same loop coordinates for the braid can be obtained without creating an intermediate loop with

```
>> loopcoords(b)

ans = (( 0  0  3 -1 -1 -1 -4  3 ))
```

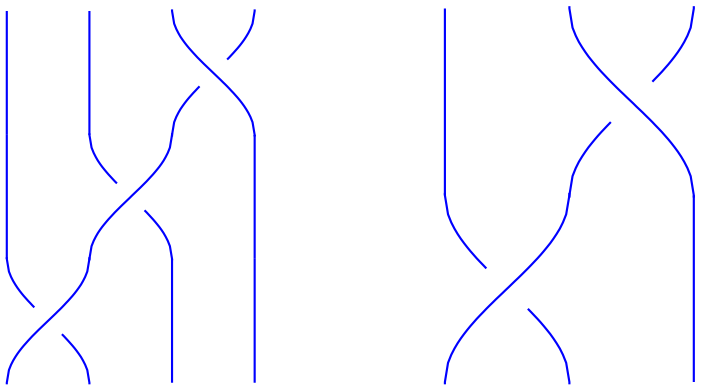
3 Side note: On filling-in punctures

Recall the command `subbraid` from Section 2.1. We took the 4-string braid $\sigma_1\sigma_2\sigma_3^{-1}$ and discarded the third string, to obtain $\sigma_1\sigma_2^{-1}$:

```
>> a = braid([1 2 -3]);
>> b = subbraid(a,[1 2 4])   % discard string 3, keep 1,2,4

b = < 1 -2 >
```

The braids `a` and `b` are shown in Fig. 5; their entropy is

```
>> a.entropy, b.entropy

ans = 0.8314
ans = 0.9624
```



(a) (b)

Figure 5: Removing the third string from the braid (a) $\sigma_1\sigma_2\sigma_3^{-1}$ yields the braid (b) $\sigma_1\sigma_2^{-1}$.

Note that the entropy of the subbraid `b` is *higher* than the original braid. This is counter-intuitive: shouldn't removing strings cause loops to shorten, therefore lowering their growth?[2]

In some sense this must be true: consider the rod-stirring device shown in Fig. 6(a), where the rods move according the to braid $\sigma_1\sigma_2\sigma_3^{-1}$. Removing the third string can be regarded as *filling-in* the third puncture (rod); clearly then the material line can be shortened, leading to a decrease in entropy.

The flaw in the argument is that even though we can remove any string, we cannot fill in a puncture that is permuted, since the resulting braid does not define a homeomorphism on the filled-in surface. To remedy this, let us take enough powers of the braid $\sigma_1\sigma_2\sigma_3^{-1}$ to ensure that the third puncture returns to its original position, using the method `perm` to find the permutation induced by the braid:

```
>> perm(a)

ans = 2     3     4     1
```

The permutation is cyclic (it can be constructed with exactly one cycle), so the fourth power should do it:

[2]In fact, the entropy obtained by the removal of a string is constrained by the minimum possible entropy for the remaining number of strings (Song *et al.*, 2002; Hironaka & Kin, 2006; Thiffeault & Finn, 2006; Ham & Song, 2007; Venzke, 2008; Lanneau & Thiffeault, 2011). So here the entropy of the 3-braid could only be zero or ≥ 0.9624.

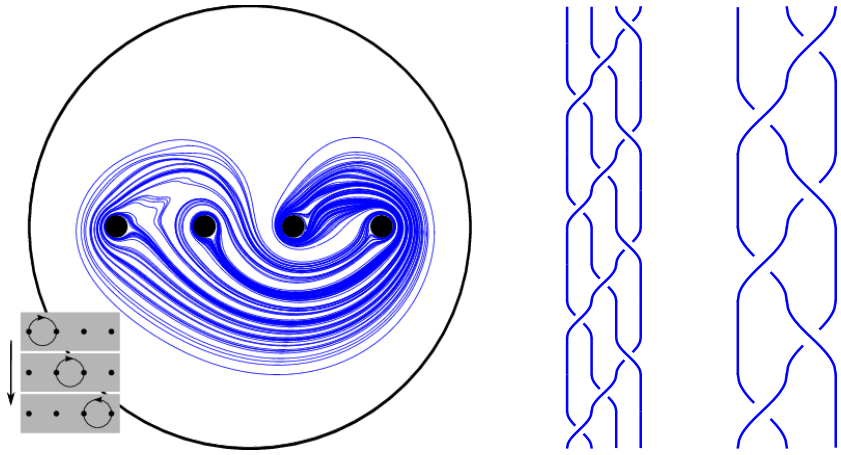


Figure 6: (a) The mixing protocol specified by the braid $\sigma_1\sigma_2\sigma_3^{-1}$ (Thiffeault *et al.*, 2008). The inset shows how the rods are moved. (b) The pure braid $(\sigma_1\sigma_2\sigma_3^{-1})^4$. (c) The braid $(\sigma_1\sigma_2^{-1})^2\sigma_1\sigma_2$, obtained by removing the third string from (b).

```
>> perm(a^4)

ans = 1     2     3     4
```

This is now a pure braid: all the strings return to their original position (Fig. 6(b)). Now here's the surprise: the subbraid obtained by removing the third string from `a^4` is

```
>> b2 = subbraid(a^4,[1 2 4])

b2 = < 1 -2  1 -2  1  2 >
```

which is *not* `b^4` (Fig. 6(c))! However, now there is no paradox in the entropies:[3]

```
>> entropy(a^4), entropy(b2)

ans = 3.3258

Warning: Failed to converge to requested tolerance; braid is
   likely finite-order or has low entropy.
```

[3]Song (2005) showed that the entropy of a pure braid is greater than 1.4436, if it is nonzero.

```
> In braid.entropy at 89

ans = 0
```

`braidlab` has trouble computing the entropy because the braid `b2` appears to be finite-order. Indeed, the braid `b2` is conjugate to σ_1^2:

```
>> c = braid([2 -1],3);
>> compact(c*b2*c^-1)

ans = < 1  1 >
```

showing that its entropy is indeed zero.

The moral is: when filling-in punctures, make sure that the strings being removed are permuted only among themselves. For very long, random braids, we still expect that removing a string will decrease the entropy, since the string being removed will have returned to its initial position many times.

Acknowledgments

The development of `braidlab` was supported by the US National Science Foundation, under grants DMS-0806821 and CMMI-1233935. The author thanks Michael Allshouse and Marko Budisic for extensive testing, comments, and for contributing some of the code. James Puckett and Karen Daniels provided the test data from their granular medium experiments (Puckett *et al.*, 2012). `braidlab` uses Toby Hall's *Train* (Hall, 2012); Jae Choon Cha's *CBraid* (Cha, 2011); Juan González-Meneses's *Braiding* (González-Meneses, 2011); John D'Errico's *Variable Precision Integer Arithmetic* (D'Errico, 2013); and Markus Buehren's *assignmentoptimal* (Buerhen, 2011).

References

Bangert, P. D., Berger, M. A. & Prandi, R. 2002 In search of minimal random braid configurations. *J. Phys. A* **35** (1), 43–59.

Bestvina, M. & Handel, M. 1995 Train-tracks for surface homeomorphisms. *Topology* **34** (1), 109–140.

Birman, J. S. 1975 *Braids, Links, and Mapping Class Groups*. *Annals of Mathematics Studies* 82. Princeton, NJ: Princeton University Press.

Birman, J. S. & Brendle, T. E. 2005 Braids: A survey. In *Handbook of Knot Theory* (ed. W. Menasco & M. Thistlethwaite), pp. 19–104. Amsterdam: Elsevier, available at `http://arXiv.org/abs/math.GT/0409205`.

Boyland, P. L. 1994 Topological methods in surface dynamics. *Topology Appl.* **58**, 223–298.

Buerhen, M. 2011 Functions for the rectangular assignment problem. `http://www.mathworks.com/matlabcentral/fileexchange/6543`.

Burau, W. 1936 Über Zopfgruppen und gleichsinnig verdrilte Verkettungen. *Abh. Math. Semin. Hamburg Univ.* **11**, 171–178.

Casson, A. J. & Bleiler, S. A. 1988 *Automorphisms of surfaces after Nielsen and Thurston*, *London Mathematical Society Student Texts*, vol. 9. Cambridge: Cambridge University Press.

Cha, J. C. 2011 *CBraid: A C++ library for computations in braid groups.* `http://code.google.com/p/cbraid`.

Dehornoy, P. 2008 Efficient solutions to the braid isotopy problem. *Discr. Applied Math.* **156**, 3091–3112.

D'Errico, J. 2013 Variable Precision Integer Arithmetic. `http://www.mathworks.com/matlabcentral/fileexchange/22725-variable-precision-integer-arithmetic`.

Dynnikov, I. A. 2002 On a Yang–Baxter map and the Dehornoy ordering. *Russian Math. Surveys* **57** (3), 592–594.

Dynnikov, I. A. & Wiest, B. 2007 On the complexity of braids. *Journal of the European Mathematical Society* **9** (4), 801–840.

Fathi, A., Laundenbach, F. & Poénaru, V. 1979 Travaux de Thurston sur les surfaces. *Astérisque* **66-67**, 1–284.

González-Meneses, J. 2011 *Braiding: A computer program for handling braids.* The version used is distributed with *CBraid*: `http://code.google.com/p/cbraid`.

Hall, T. 2012 *Train: A C++ program for computing train tracks of surface homeomorphisms.* `http://www.liv.ac.uk/~tobyhall/T_Hall.html`.

Hall, T. & Yurttaş, S. Ö. 2009 On the topological entropy of families of braids. *Topology Appl.* **156** (8), 1554–1564.

Ham, J.-Y. & Song, W. T. 2007 The minimum dilatation of pseudo-Anosov 5-braids. *Experiment. Math.* **16** (2), 167–179.

Hironaka, E. & Kin, E. 2006 A family of pseudo-Anosov braids with small dilatation. *Algebraic & Geometric Topology* **6**, 699–738.

Kassel, C. & Turaev, V. 2008 *Braid groups*. New York, NY: Springer.

Lanneau, E. & Thiffeault, J.-L. 2011 On the minimum dilatation of braids on the punctured disc. *Geometriae Dedicata* **152** (1), 165–182.

Moussafir, J.-O. 2006 On computing the entropy of braids. *Func. Anal. and Other Math.* **1** (1), 37–46.

Paterson, M. S. & Razborov, A. A. 1991 The set of minimal braids is co-NP complete. *J. Algorithm* **12**, 393–408.

Puckett, J. G., Lechenault, F., Daniels, K. E. & Thiffeault, J.-L. 2012 Trajectory entanglement in dense granular materials. *Journal of Statistical Mechanics: Theory and Experiment* **2012** (6), P06008.

Song, W. T. 2005 Upper and lower bounds for the minimal positive entropy of pure braids. *Bull. London Math. Soc.* **37** (2), 224–229.

Song, W. T., Ko, K. H. & Los, J. E. 2002 Entropies of braids. *J. Knot Th. Ramifications* **11** (4), 647–666.

Thiffeault, J.-L. 2005 Measuring topological chaos. *Phys. Rev. Lett.* **94** (8), 084502.

Thiffeault, J.-L. 2010 Braids of entangled particle trajectories. *Chaos* **20**, 017516.

Thiffeault, J.-L. & Finn, M. D. 2006 Topology, braids, and mixing in fluids. *Phil. Trans. R. Soc. Lond. A* **364**, 3251–3266.

Thiffeault, J.-L., Finn, M. D., Gouillart, E. & Hall, T. 2008 Topology of chaotic mixing patterns. *Chaos* **18**, 033123.

Thurston, W. P. 1988 On the geometry and dynamics of diffeomorphisms of surfaces. *Bull. Am. Math. Soc.* **19**, 417–431.

Venzke, R. W. 2008 Braid forcing, hyperbolic geometry, and pseudo-Anosov sequences of low entropy. PhD thesis, California Institute of Technology.

Weisstein, E. W. 2013 Alexander polynomial. From *MathWorld*—A Wolfram Web Resource. `http://mathworld.wolfram.com/AlexanderPolynomial.html`.

Index

Alexander–Conway polynomial, 7–8

Bestvina–Handel algorithm, 6
braid
  Burau representation, 7–8
  closure, 9–11
  complexity, 6–7, 14
  conjugate, 9–11
  entropy, 5–6, 15–19
    minimum, 17
  finite-order, 6, 18–19
  from data, 8–11
  Garside form, 11
  loop coordinates, 15–16
  pseudo-Anosov, 6
  pure, 8, 18
  reducible, 6
  Thurston–Nielsen type, 6
  writhe, 8
`braid` class
  `compact`, 19
`braid` class, 3–11
  action on `loop` (`*`), 13–15
  `alexpoly`, 7–8
  `burau`, 7–8
  `closure`, 10–11
  `compact`, 4, 9–10
  `complexity`, 6–7, 14
  `conjtest`, 10–11
  constructor, 3–5
    from data, 8–11
    half-twist, 5
    random braid, 4
  `entropy`, 5–6, 15–19
  equality (`==`), 5
  identity braid, 4
  inverse (`inv`), 3
  `ispure`, 8
  `istrivial`, 4
  `length`, 9
  `lexeq`, 5
  `loopcoords`, 16
  multiplication (`*`), 3, 13–15
  number of strings (`n`), 4
  `perm`, 8, 17–19
  `plot`, 10
  power (`^`), 3
  `subbraid`, 5, 10, 16
  `tensor`, 5
  `tntype`, 6
  `writhe`, 8

CBraid, 11
complexity, *see* braid complexity
crossings, 8, 9, *see also* projection line
  in braid closure, 10–11

disk, punctured, 5, 11, 15
Dynnikov coordinates, *see* loop coordinates

entropy, *see* braid entropy

geometric complexity, *see* braid complexity

`help`, 4
homeomorphism, 5
  isotopy classes, 5

installing `braidlab`, 2
integral lamination, *see* loop, multi-
intersection numbers, *see* loop intersection numbers

Laurent polynomials, 7–8
`laurpoly`, 7–8
loop
    coordinates, 5, 11–16
    essential, 11
    homotopy classes, 11
    intersection numbers, 11–13
    minimum length, 14
    multi-, 12, 15–16
`loop` class, 11–16
    `a`, `b`, `ab`, 13
    constructor, 12, 14–15
    `intaxis`, 14
    `minlength`, 14
    number of punctures (`n`), 13
    `plot`, 12
    vectorized, 14–15

Matlab
    MEX files, 2
    namespace, 2
    path, 2
    wavelet toolbox, 7
Mercurial, 2
multiloop, *see* loop, multi-

projection line, 8–11, *see also* crossings
punctures, 5, 11, 13–15
    filling-in, 16–19

testsuite, 3, 8
topological entropy, *see* braid entropy