

# braidlab user's guide

Jean-Luc Thiffeault

release 1.0.5

## Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Installing braidlab</b>                                   | <b>2</b>  |
| 1.1      | Precompiled packages . . . . .                               | 2         |
| 1.2      | Cloning the repository . . . . .                             | 2         |
| 1.3      | Setting Matlab's path . . . . .                              | 2         |
| 1.4      | Testing your installation . . . . .                          | 3         |
| <b>2</b> | <b>A tour of braidlab</b>                                    | <b>3</b>  |
| 2.1      | The <code>braid</code> class . . . . .                       | 3         |
| 2.1.1    | Constructor and elementary operations . . . . .              | 3         |
| 2.1.2    | Topological entropy and complexity . . . . .                 | 6         |
| 2.1.3    | Representation and invariants . . . . .                      | 7         |
| 2.2      | Constructing a braid from data . . . . .                     | 10        |
| 2.2.1    | An example . . . . .   | 10        |
| 2.2.2    | Changing the projection line and enforcing closure . . . . . | 11        |
| 2.2.3    | The <code>databraid</code> subclass . . . . .                | 13        |
| 2.3      | The <code>loop</code> class . . . . .                        | 14        |
| 2.3.1    | Loop coordinates . . . . .                                   | 14        |
| 2.3.2    | Acting on loops with braids . . . . .                        | 15        |
| 2.4      | Loop coordinates for a braid . . . . .                       | 17        |
| <b>3</b> | <b>An example: Taffy pullers</b>                             | <b>19</b> |
| <b>4</b> | <b>Side note: On filling-in punctures</b>                    | <b>22</b> |
|          | <b>Acknowledgments</b>                                       | <b>25</b> |

|            |    |
|------------|----|
| References | 27 |
| Index      | 28 |

## 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
```

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 (much) more slowly.

If GMP (the GNU MultiPrecision library) is installed on your system, instead of the above you can compile braidlab with

```
# cd braidlab; make BRAIDLAB_USE_GMP=1
```

This will speed up some functions, in particular testing for equality of large braids.

### 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 >

>> braid('8_21') % braid for 8-crossing knot #21

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

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

The `braid` class 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. Returning zero
entropy.

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 (Dyannikov & 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 = bureau(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)
```

Another option is to use Matlab's symbolic toolbox:

```
>> B = bureau(braid([1 -2]), sym('t'))
```

```

B = [ -t,      t]
     [-1, 1 - 1/t]

```

where now B is a matrix of `sym` objects:

```
>> B(2,2)
```

```
ans = 1 - 1/t
```

The reduced Burau matrix of a braid can be used to compute the *Alexander-Conway polynomial* (or Alexander polynomial for short) of its closure. For instance, the trefoil knot is given by the closure of the braid  $\sigma_1^3$  (Weisstein, 2013), which gives a Laurent polynomial

```
>> alexpoly(braid([1 1 1])) % can also use braid('trefoil')
```

```
ans(z) = + z^(+2) - z^(+1) + 1
```

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

```
>> alexpoly(braid([1 -2 1 -2])) % or braid('figure-8')
```

```
ans(z) = - 1 + 3*z^(-1) - z^(-2)
```

This can be 'centered' so that it satisfies  $p(z) = \pm p(1/z)$ :



```
>> alexpoly(braid([1 -2 1 -2]),'centered')
```

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

The centered Alexander polynomial is a knot invariant, so it can be used to determine when two knots are not the same. For knots, the centered polynomial is guaranteed to have integral powers. For links, such as the Hopf link consisting of two singly-linked loops, it might not:

```
>> alexpoly(braid([1 1]),'centered') % the Hopf link
```

```
Error using braidlab.braid/alexpoly
Polynomial with fractional powers. Remove 'centered' option
or use the symbolic toolbox.
```

Fractional powers cannot be represented with a `laurpoly` object. In that case we can drop the ‘centered’ option, which yields the uncentered polynomial  $1 - z$ . Alternatively, we can switch to using a variable from the symbolic toolbox:

```
>> alexpoly(braid([1 1]),sym('x'),'centered')
```

```
ans = 1/x^(1/2) - x^(1/2)
```

which can represent fractional powers. This polynomial satisfies  $p(x) = -p(1/x)$ .

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

### 2.2.1 An example

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 |
|------|----------|--------|--------|------------|
| 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

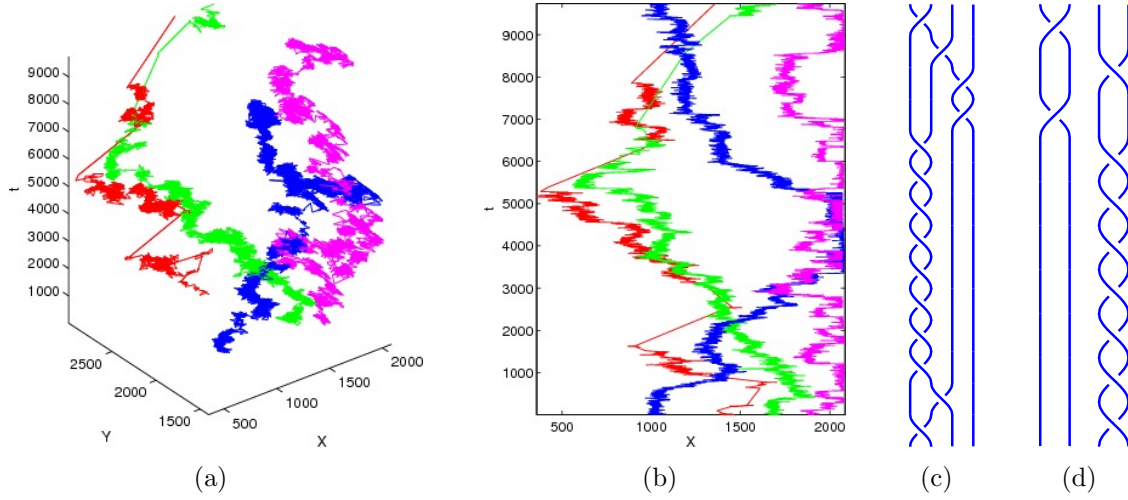


Figure 1: (a) A dataset of four trajectories, (b) projected along the  $X$  axis. (c) The compacted braid  $\sigma_1^{-1}\sigma_2^{-1}\sigma_1^{-8}\sigma_3^2\sigma_2\sigma_1$  corresponding to the  $X$  projection in (b). (d) The compacted braid  $\sigma_3^{-7}\sigma_1\sigma_3^{-1}\sigma_1$  corresponding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

braid diagram allows us to see some 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  $-4$ .

### 2.2.2 Changing the projection line and enforcing closure

The braid in the previous section 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 (Boylan, 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  $\mathbf{bX}$  is unchanged by the closure, but here the  $Y$ -projected braid  $\mathbf{bY}$  is longer by one generator ( $\mathbf{bY}$  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  $\mathbf{c}$  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.

### 2.2.3 The databraid subclass

In some instances when dealing with data it is important to know the *crossing times*, that is, the times at which two particles exchanged position along the projection line. A braid object does not keep this information, but there is an object that does: a **databraid**. Its constructor takes an optional vector of times as an argument, and it has a data member `tcross` that retains the crossing times. Using the same data `XY` from before, sampled at times `ti`, we have

```
>> b = databraid(XY,ti);
>> b.tcross(1:3)

ans = 870.9010
      872.1758
      887.0089
```

There are always exactly as many crossing times as generators in the braid. Many operations that can be done to a **braid** also work on a **databraid**, with a few differences:

- `compact` works a bit differently. It is less effective than `braid.compact` since it must preserve the order of generators in order to maintain the ordering of the crossing times.
- Equality testing checks if two **databraids** are lexicographically equal (i.e., generator-by-generator) and that their crossing times all agree. This is very restrictive. To check if the underlying braids are equal, first convert the **databraids** to **braids** by using the method `databraid.braid`.
- Multiplication of two **databraids** is only defined if the crossing times of the first braid are all earlier than the second.
- Powers and inverses of **databraids** are not defined.

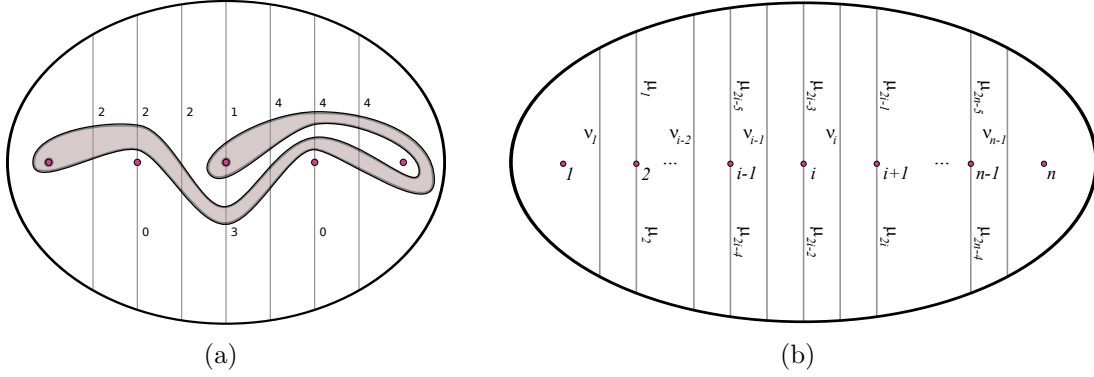


Figure 2: (a) A simple close loop in a disk with  $n = 5$  punctures. (b) Definition of intersection numbers  $\mu_i$  and  $\nu_i$ . [From Thiffeault (2010).]

## 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  $\mu_i$  and  $\nu_i$  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 more convenient to define

$$a_i = \frac{1}{2} (\mu_{2i} - \mu_{2i-1}), \quad b_i = \frac{1}{2} (\nu_i - \nu_{i+1}), \quad i = 1, \dots, n-2. \quad (1)$$

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

$$\mathbf{u} = (a_1, \dots, a_{n-2}, b_1, \dots, b_{n-2}), \quad (2)$$

which gives the *loop coordinates* (or *Dynnikov coordinates*) for the loop. (Some authors such as Dehornoy (2008) give the coordinates as  $(a_1, b_1, \dots, a_{n-2}, b_{n-2})$ .) There is now a bijection between  $\mathbb{Z}^{2n-4}$  and essential simple closed loops (Dynnikov,

2002; Moussafir, 2006; Hall & Yurttas, 2009; Thiffeault, 2010). Actually, *multiloops*: loop coordinates can describe unions of disjoint loops (see Section 2.4).<sup>1</sup>

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]
```

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  $\sigma_1^{-1}$  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
```

---

<sup>1</sup>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).

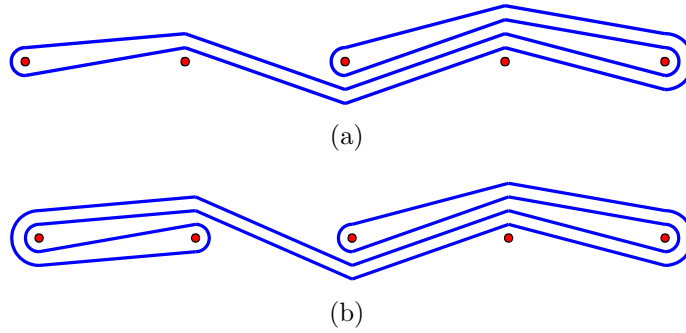


Figure 3: (a) The loop  $((-1 \ 1 \ -2 \ 0 \ -1 \ 0))$ . (b) The braid generator  $\sigma_1^{-1}$  applied to the loop in (a).

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

Figure 3(b) shows `plot(b*1)`. The first and second punctures were interchanged counterclockwise (the action of  $\sigma_1^{-1}$ ), 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(1)
```

```
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(1)
```

```
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:

```
>> l1 = loop([-1 1 -2 0; 1 -2 3 4])
```



```
l1 = (( -1  1 -2  0 ))
      ((  1 -2  3  4 ))
```

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

```
>> minlength(l1)

ans = 14
      34
```

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

```
>> b = braid([1 -2]);
>> b*l1

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(l1(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)
```

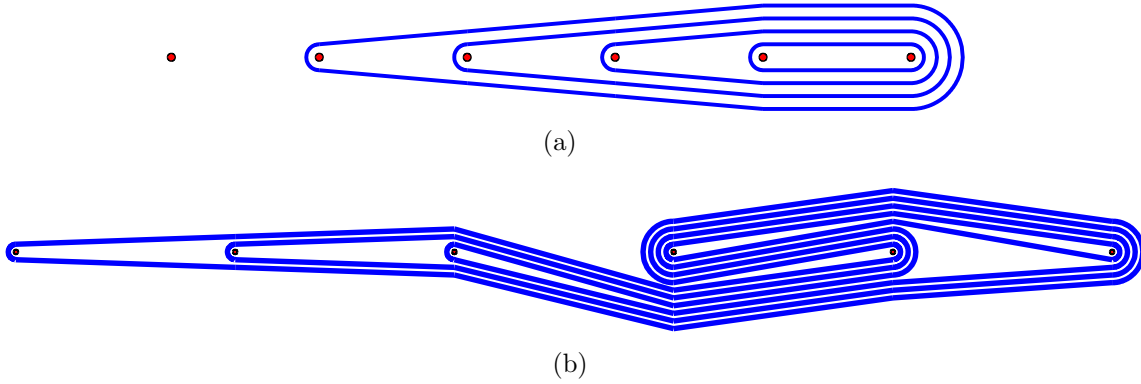


Figure 4: (a) The multiloop created by `loop(5)`. (b) The multiloop `b*loop(5)`, where `b` is the braid  $\sigma_1 \sigma_2 \sigma_3 \sigma_4^{-1}$ .

```
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 ))
```

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 ))
```

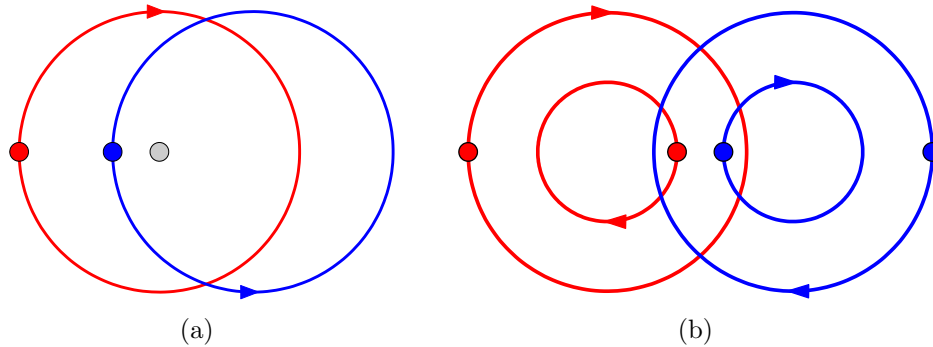


Figure 5: (a) Three-rod taffy puller. (b) Four-rod taffy puller.

### 3 An example: Taffy pullers

Taffy pullers are a class of devices designed to stretch and fold soft candy repeatedly (Finn & Thiffeault, 2011). The goal is to aerate the taffy. Since many folds are required, the process has been mechanized using fixed and moving rods. The two most typical designs are shown in Figure 5: the one in Figure 5(a) has a single fixed rod (gray) and two moving rods, each rotating on a different axis. The design in Figure 5(b) has four moving rods, sharing two axes of rotation. (There are several videos of taffy pullers on YouTube.)

Let's use `braidlab` to analyze the rod motion. From the folder `doc/examples`, run the command

```
>> b = taffy('3rods')

b = < -2  1  1 -2 >
```

which also produces Figure 5(a). The Thurston–Nielsen type and topological entropy of this braid are

```
>> [t,entr] = tntype(b)

t = pseudo-Anosov

entr = 1.7627
```

One would expect a competent taffy puller to be pseudo-Anosov, as this one is. It implies that there is no ‘bad’ initial condition where a piece of taffy never gets stretched, or stretches slowly. A reducible or finite-order braid would indicate poor

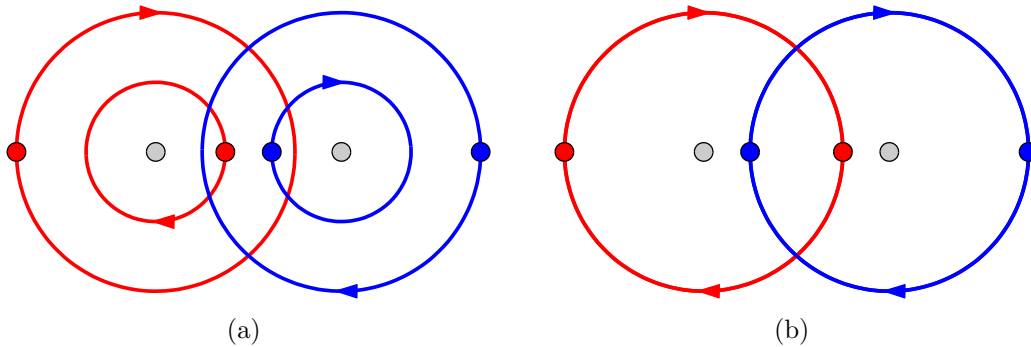


Figure 6: (a) A six-rod taffy puller based on Figure 5(b), with two added fixed rods (gray). This is a poor design, since it leads to a reducible braid. (b) Same as (a), but with the same radius of motion for all the rods. The braid is in this case pseudo-Anosov, with larger entropy than the 4-rod design.

design. The entropy is a measure of the taffy puller's effectiveness: it gives the rate of growth of curves anchored on the rods. Thus, the length of the taffy is multiplied (asymptotically) by  $e^{1.7627} \simeq 5.828$  for each full period of rod motion. Needless to say, this leads to extremely rapid growth, since after 10 periods the taffy length has been multiplied by roughly  $10^7$ .

The design in Figure 5(b) can be plotted and analyzed with

```
>> b = taffy('4rods')
b = < 1 3 2 2 1 3 >
```

When we apply `tn_type` to this braid we find the braid is pseudo-Anosov with exactly the same entropy as the 3-rod taffy puller, 1.7627. There is thus no obvious advantage to using more rods in this case.

A simple modification of the 4-rod design in Figure 5(b) is shown in Figure 6(a). The only change is to extend the rotation axes into two extra fixed rods (shown in gray). The resulting braid is

```
>> b = taffy('6rods-bad')
b = < 2 1 2 4 5 4 3 3 2 1 2 4 5 4 >
```

with Thurston–Nielsen type

```
>> tn_type(b)
```

```
ans = reducible
```

There are reducing curves in this design: simply wrap a loop around the left gray rod and the inner red rod, and it will rotate without stretching. To avoid this, we extend the radius of motion of the inner rods to equal that of the outer ones, and obtain the design shown in Figure 5(b). The corresponding braid is

```
>> b = taffy('6rods')

b = < 3  2  1  2  4  5  4  3  3  2  1  2  5  4  5  3 >
```

with Thurston–Nielsen type and entropy

```
>> [t,entr] = tntype(b)

t = pseudo-Anosov

entr = 2.6339
```

The fixed rods have increased the entropy by 50%! This sounds like a fairly small change, but what it means is that this 6-rod design achieves growth of  $10^7$  in about 6 iterations rather than 10. Alexander Flanagan constructed this six-rod device while an undergraduate student at the University of Wisconsin – Madison, but as far as we know this new design has not yet been used in commercial applications.

The symmetric design of the taffy pullers illustrates one pitfall when constructing braids. If we give an optional projection angle of  $\pi/2$  to **taffy**:

```
>> taffy('4rods',pi/2)
Error using colorbraiding
Coincident projection coordinate; change projection angle
(type help braid.braid).
```

This corresponds to using the  $y$  (vertical) axis to compute the braid, but as we can see from Figure 5(b) this is a bad choice, since all the rods are initially perfectly aligned along that axis. The braid obtained would depend sensitively on numerical roundoff when comparing the rod projections. Instead of attempting to construct the braid, **braidlab** returns an error and asks the user to modify the projection axis. A tiny change in the projection line is sufficient to break the symmetry:

```
>> taffy('4rods',pi/2 + .01)

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

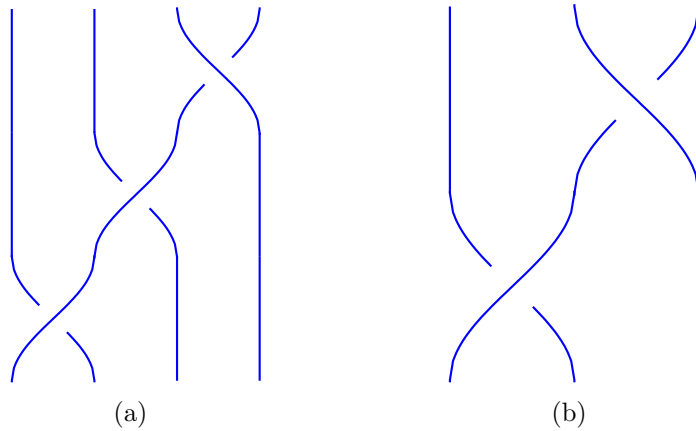


Figure 7: Removing the third string from the braid (a)  $\sigma_1 \sigma_2 \sigma_3^{-1}$  yields the braid (b)  $\sigma_1 \sigma_2^{-1}$ .

```
>> compact(ans)

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

which is actually equal to the braid formed from projecting on the  $x$  axis, though it need only be conjugate (see Section 2.2).

## 4 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. 7; their entropy is

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

ans = 0.8314
ans = 0.9624
```

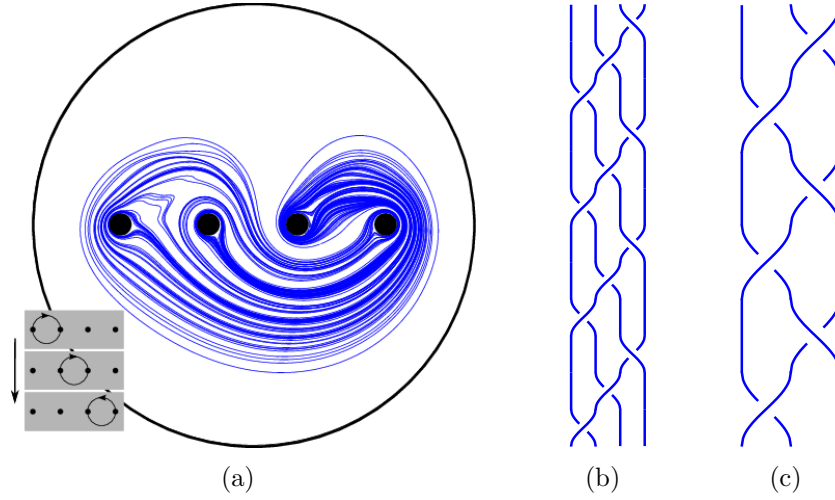


Figure 8: (a) The mixing protocol specified by the braid  $\sigma_1\sigma_2\sigma_3^{-1}$  (Thiffeault *et al.*, 2008). The inset shows how the rods are moved. (b) The pure braid  $(\sigma_1\sigma_2\sigma_3^{-1})^4$ . (c) The braid  $(\sigma_1\sigma_2^{-1})^2\sigma_1\sigma_2$ , obtained by removing the third string from (b).

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?<sup>2</sup>

In some sense this must be true: consider the rod-stirring device shown in Fig. 8(a), where the rods move according to the 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
```

<sup>2</sup>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  $\geq 0.9624$ .

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

```
>> perm(a^4)

ans = 1      2      3      4
```

This is now a pure braid: all the strings return to their original position (Fig. 8(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. 8(c))! However, now there is no paradox in the entropies:<sup>3</sup>

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

ans = 3.3258

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

ans = 0
```

`braidlab` has trouble computing the entropy because the braid `b2` appears to be finite-order. Indeed, the braid `b2` is conjugate to  $\sigma_1^2$ :

```
>> 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.

---

<sup>3</sup>Song (2005) showed that the entropy of a pure braid is greater than 1.4436, if it is nonzero.



# 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); Markus Buehnen’s *assignmentoptimal* (Buehnen, 2011); and Jakob Progsch’s *ThreadPool* (Progsch, 2012).

# 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 verdrehte 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.
- FINN, M. D. & THIFFEAULT, J.-L. 2011 Topological optimization of rod-stirring devices. *SIAM Rev.* **53** (4), 723–743.
- 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](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.

- PROGSCH, J. 2012 Threadpool: A simple C++11 thread pool implementation. <https://github.com/progschj/ThreadPool>.
- 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, 8–9

Bestvina–Handel algorithm, 6–7

braid

    Bureau representation, 7–9

    closure, 10, 12

    complexity, 7, 16

    conjugate, 10–12, 21

    entropy, 6–7, 17, 19, 22–24

        minimum, 22

    finite-order, 6–7, 24

    from data, 9–13

    Garside form, 12

    loop coordinates, 17–18

    pseudo-Anosov, 6–7, 19

    pure, 9, 23, 24

    reducible, 6–7, 20

    Thurston–Nielsen type, 6, 19

    writhe, 9

**braid class**, 3–13

    action on loop (\*), 15–17

    alexpoly, 8–9

    bureau, 7–9

    closure, 12

    compact, 4, 11–13, 24

    complexity, 7, 16

    conjtest, 11–12

    constructor, 3–5

        from data, 9–13

        half-twist, 5

        knots, 5

        random braid, 5

    entropy, 6–7, 17, 22–24

    equality (==), 5

    identity braid, 4

    inverse (inv), 4

    ispure, 9

    istrivial, 4

    length, 10

    lexeq, 5

    loopcoords, 18

    multiplication (\*), 3, 15–17

    number of strings (n), 4

    perm, 9, 23–24

    plot, 11

    power (^), 4

    subbraid, 5, 11, 22

    tensor, 5

    tntype, 6–7, 19–21

    writhe, 9

CBraid, 12

complexity, *see* braid complexity

crossing, 9, 11, *see also* projection line

    in braid closure, 12

    times, 12–13

**databraid class**, 13

disk, punctured, 6, 13, 17

Dynnikov coordinates, *see* loop coordinates

entropy, *see* braid entropy

geometric complexity, *see* braid complexity

GMP, 2

help, 4–5

homeomorphism, 6

    isotopy classes, 6

installing **braidlab**, 2

- integral lamination, *see* loop, multi-
- intersection numbers, *see* loop intersection numbers
- knot
  - Alexander polynomial, 8
  - braid representative, 5
  - figure-eight, 8
  - invariant, 8
  - trefoil, 8
- Laurent polynomials, 7–9
- `laurpoly`, 7–9
- loop
  - coordinates, 5, 14–15, 17–18
  - essential, 13
  - homotopy classes, 13
  - intersection numbers, 13–15
  - minimum length, 15
  - multi-, 14, 17–18
- loop class, 13–18
  - `a`, `b`, `ab`, 15
  - constructor, 14, 16–17
  - `intaxis`, 16
  - `minlength`, 15–16
  - number of punctures (`n`), 15
  - `plot`, 14
  - vectorized, 16–17
- Matlab
  - MEX files, 2
  - namespace, 2
  - path, 2
  - symbolic toolbox, 8, 9
  - wavelet toolbox, 7
- Mercurial, 2
- multiloop, *see* loop, multi-
- projection line, 9–12, *see also* crossing
  - bad choice of angle, 21
- punctures, 6, 13, 15, 17
  - filling-in, 22–24
- `sym`, 9
- `taffy`, 19–21
- taffy pullers, 18–21
- `tcross`, 13
- testsuite, 3, 9
- topological entropy, *see* braid entropy