The braids package: codebase

Andrew Stacey
loopspace@mathforge.org

v2.0 from 2019/03/20

1 Introduction

This is a package for drawing braid diagrams using PGF/TikZ. Its inspiration was a question and answer on the website http://tex.stackexchange.com.

2 History

- v1.0 First public release.
- v1.1 Added ability to configure the gap size, the control points, and the "nudge". Added ability to add labels to strands between crossings.
- v2 Reimplemented as TikZ library rather than a standalone package.

3 Implementation

Issue a notice that this is a depreciated version of the braids package.

```
\PackageWarning{braids}{%
  This package has been reimplemented as a TikZ library; if starting with a fresh document,
}%
```

\ge@addto@macro This is an expanded version of \g@addto@macro. Namely, it adds the *expansion* of the second argument to the first.

```
\long\def\ge@addto@macro#1#2{%
  \begingroup
  \toks@\expandafter\expandafter\expandafter{\expandafter#1#2}%
  \xdef#1{\the\toks@}%
  \endgroup}
```

(*End definition for* \ge@addto@macro.)

\braid This is the user command. We start a group to ensure that all our assignments are local, and then call our initialisation code. The optional argument is for any keys to be set.

```
\newcommand{\braid}[1][]{%
  \begingroup
  \braid@start{#1}}
```

(*End definition for* \braid. *This function is documented on page* ??.)

\braid@process This is the token swallower. This takes the next token on the braid specification and passes it to the handler command (in the macro `\braid@token`) which decides what to do next. (Incidentally, the code here is heavily influenced by TikZ. That's probably not very surprising.)

```
\def\braid@process{%
  \afterassignment\braid@handle\let\braid@token=%
}
```

(*End definition for* `\braid@process`.)

\braid@process@start This is a variant of `\braid@process` which is used at the start where we might have a few extra bits and pieces before the braid itself starts. Specifically, we test for the `at` and `(name)` possibilities.

```
\def\braid@process@start{%
  \afterassignment\braid@handle@start\let\braid@token=%
}
```

(*End definition for* `\braid@process@start`.)

\braid@handle@start This is the handler in use at the start. It looks for the tokens `a` or `(` which (might) signal the start of an `at (coordinate)` or `(name)`. If we get anything else (modulo spaces) we decide that we've reached the end of the initialisation stuff and it is time to get started on the braid itself.

```
\def\braid@handle@start{%
  \let\braid@next=\braid@handle
  \ifx\braid@token a
```

We got an `a` so we might have an `at (coordinate)`

```
   \let\braid@next=\braid@maybe@locate
  \else
  \ifx\braid@token(%)
```

We got an `(` so we have a name

```
   \iffalse)\fi %Indentation hack!
   \let\braid@next=\braid@assign@name
  \else
  \ifx\braid@token\@sptoken
```

Space; boring, redo from start

```
   \let\braid@next=\braid@process@start
  \fi
  \fi
  \fi
  \braid@next%
}
```

(*End definition for* `\braid@handle@start`.)

\braid@handle This is the main handler for parsing the braid word. It decides what action to take depending on what the token is. We have to be a bit careful with catcodes, some packages set `;` and `|` to be active. We should probably also be careful with `^` and `_`.

```
\let\braid@semicolon=;
\let\braid@bar=|
\def\braid@handle{%
  \let\braid@next=\braid@process
```

Start by checking our catcodes to see what we should check against

```
\ifnum\the\catcode`\;=\active
\expandafter\let\expandafter\braid@semicolon\tikz@activesemicolon
\fi
\ifnum\the\catcode`\|=\active
\expandafter\let\expandafter\braid@bar\tikz@activebar
\fi
\ifx\braid@token\braid@semicolon
```

Semicolon, means that we're done reading our braid. It's time to render it.

```
 \let\braid@next=\braid@render
\else
\ifx\braid@token^
```

Superscript character, the next token tells us whether it's an over-crossing or an under-crossing.

```
 \let\braid@next=\braid@sup
\else
\ifx\braid@token_
```

Subscript character, the next token tells us which strands cross.

```
 \let\braid@next=\braid@sub
\else
\ifx\braid@token-
```

Hyphen, this is so that we can have more than one crossing on the same level.

```
 \braid@increase@levelfalse
\else
\ifx\braid@token1%
```

1: this means the "identity" crossing, so no crossing here. Increase the level, unless overriden, and add to the label.

```
 \ifbraid@increase@level
  \stepcounter{braid@level}
 \fi
 \braid@increase@leveltrue
 \ge@addto@macro\braid@label{\braid@token}%
\else
\ifx\braid@token[%
```

Open bracket, this means we have some more options to process.

```
 \let\braid@next=\braid@process@options
\else
\ifx\braid@token\braid@bar
```

Bar, this tells us that we want a "floor" at this point.

```
 \edef\braid@tmp{,\expandafter\the\value{braid@level}}%
 \ge@addto@macro\braid@floors\braid@tmp%
\else
\ifx\braid@token\bgroup
```

Begin group, which we reinterpret as begining a scope.

```
 \braid@beginscope
\else
\ifx\braid@token\egroup
```

End group, which ends the scope

```
    \braid@endscope
  \else
  \ifx\braid@token\braid@olabel@strand
   \let\braid@next=\braid@olabel@strand
  \else
  \ifx\braid@token\braid@clabel@strand
   \let\braid@next=\braid@clabel@strand
  \else
```

Otherwise, we add the token to the braid label.

```
  \ge@addto@macro\braid@label{\braid@token}%
  \fi
  \fi
  \fi
  \fi
  \fi
  \fi
  \fi
  \fi
  \fi
  \fi
  \braid@next%
}
```

(*End definition for* `\braid@handle`.)

`\braid@maybe@locate` If we got an `a` token in the `\braid@handle@start` then it *might* mean we're looking at `at (coordinate)` or it might mean that the user has decided to use `a` as the braid parameter. So we examine the next token for a `t`.

```
\def\braid@maybe@locate{%
  \afterassignment\braid@@maybe@locate\let\braid@token=%
}
```

(*End definition for* `\braid@maybe@locate`.)

`\braid@@maybe@locate` This is where we test for `t` and act appropriately.

```
\def\braid@@maybe@locate{%
  \let\braid@next=\braid@handle
  \ifx\braid@token t
   \let\braid@next=\braid@find@location
  \fi
  \braid@next%
}
```

(*End definition for* `\braid@@maybe@locate`.)

`\braid@find@location` This macro starts us looking for a coordinate.

```
\def\braid@find@location{%
  \afterassignment\braid@@find@location\let\braid@token=%
}
```

(*End definition for* `\braid@find@location`.)

\braid@@find@location This is the test for the start of a coordinate. If we get a (that means we've reached the coordinate. A space means "carry on". Anything else is a (non-fatal) error.

```
\def\braid@@find@location{%
  \let\braid@next=\braid@location@error
  \ifx\braid@token(%)
   \let\braid@next=\braid@locate
  \else
  \ifx\braid@token\@sptoken
   \let\braid@next=\braid@find@location
  \fi
  \fi
  \braid@next%
}
```

(*End definition for* \braid@@find@location.)

\braid@location@error This is our error message for not getting a location.

```
\def\braid@location@error{%
  \PackageWarning{braids}{Could not figure out location for braid}%
  \braid@process@start%
}
```

(*End definition for* \braid@location@error.)

\braid@locate If we reached a (when looking for a coordinate, everything up to the next) is that coordinate. Then we parse the coordinate and call the relocation macro.

```
\def\braid@locate#1){%
  \tikz@scan@one@point\braid@relocate(#1)%
}
```

(*End definition for* \braid@locate.)

\braid@relocate This is the macro that actually does the relocation.

```
\def\braid@relocate#1{%
  #1\relax
  \advance\pgf@x by -\braid@width
  \pgftransformshift{\pgfqpoint{\pgf@x}{\pgf@y}}
  \braid@process@start%
}
```

(*End definition for* \braid@relocate.)

\braid@assign@name This macro saves our name.

```
\def\braid@assign@name#1){%
  \def\braid@name{#1}%
  \braid@process@start%
}
```

(*End definition for* \braid@assign@name.)

\braid@process@options The intention of this macro is to allow setting of style options mid-braid. (At present, this wouldn't make a lot of sense.)

```
\def\braid@process@options#1]{%
    \tikzset{#1}%
  \braid@process%
}
```

(*End definition for* `\braid@process@options`.)

The next macros handle the actual braid elements. Everything has to have a subscript, but the superscript is optional and can come before or after the subscript.

`\braid@sup` This handles braid elements of the form `a^{-1}_2`.

```
\def\braid@sup#1_#2{%
  \g@addto@macro\braid@label{_{#2}^{#1}}%
  \braid@add@crossing{#2}{#1}%
}
```

(*End definition for* `\braid@sup`.)

`\braid@sub`

```
% This handles braid elements of the form \Verb+a_1+ or \Verb+a_1^{-1}+.
\def\braid@sub#1{%
  \@ifnextchar^{\braid@@sub{#1}}%
  {\g@addto@macro\braid@label{_{#1}}\braid@add@crossing{#1}{1}}%
}
```

(*End definition for* `\braid@sub`.)

`\braid@@sub` Helper macro for `\braid@sub`.

```
\def\braid@@sub#1^#2{%
  \g@addto@macro\braid@label{_{#1}^{#2}}%
  \braid@add@crossing{#1}{#2}%
}
```

(*End definition for* `\braid@@sub`.)

`\braid@ne` Remember what `1` looks like for testing against.

```
\def\braid@ne{1}
```

(*End definition for* `\braid@ne`.)

`\braid@add@crossing` This is the macro which adds the crossing to the current list of strands. The strands are stored as *soft paths* (see the TikZ/PGF documentation). So this selects the right strands and then extends them according to the crossing type.

```
\def\braid@add@crossing#1#2{%
```

Our crossing type, which is `#2`, is one of `1` or `-1`. Our strands are `#1` and `#1+1`.

```
  \edef\braid@crossing@type{#2}%
  \edef\braid@this@strand{#1}%
  \pgfmathtruncatemacro{\braid@next@strand}{#1+1}
```

Increment the level counter, if requested. The controls whether the crossing is on the same level as the previous one or is one level further on.

```
  \ifbraid@increase@level
  \stepcounter{braid@level}
  \fi
```

Default is to request increment so we set it for next time.

```
  \braid@increase@leveltrue
```

Now we figure out the coordinates of the crossing. `(\braid@tx,\braid@ty)` is the top-left corner (assuming the braid flows down the page). `(\braid@nx,\braid@ny)` is the bottom-right corner (assuming the braid flows down the page). We start by setting `(\braid@tx,\braid@ty)` according to the level and strand number, then shift `\braid@ty` by `\braid@eh` which is the "edge height" (the little extra at the start and end of each strand). Then from these values, we set `(\braid@nx,\braid@ny)` by adding on the appropriate amount. The heights `\braid@cy` and `\braid@dy` are for the control points for the strands as they cross. They're actually the same height, but using two gives us the possibility of changing them independently in a later version of this package. Lastly, we bring `\braid@ty` and `\braid@ny` towards each other just a little so that there is "clear water" between subsequent crossings (makes it look a bit better if the same strand is used in subsequent crossings).

```
\braid@tx=\braid@this@strand\braid@width
\braid@ty=\value{braid@level}\braid@height
\advance\braid@ty by \braid@eh
\braid@nx=\braid@tx
\braid@ny=\braid@ty
\advance\braid@nx by \braid@width
\advance\braid@ny by \braid@height
\advance\braid@ty by \braid@nf\braid@height
\advance\braid@ny by -\braid@nf\braid@height
\braid@cy=\braid@ty
\braid@dy=\braid@ny
\advance\braid@cy by \braid@cf\braid@height
\advance\braid@dy by -\braid@cf\braid@height
```

Now we try to find a starting point for the strand ending here. We might not have used this strand before, so it might not exist.

```
\expandafter\let\expandafter\braid@this@path@origin%
\csname braid@strand@\braid@this@strand @origin\endcsname
```

If we haven't seen this strand before, that one will be `\relax`.

```
\ifx\braid@this@path@origin\relax
```

Haven't seen this strand before, so initialise it. Record the initial position of the strand.

```
\let\braid@this@path@origin\braid@this@strand
```

Start a new soft path.

```
\pgfsyssoftpath@setcurrentpath{\@empty}
\pgfpathmoveto{\pgfpoint{\braid@tx}{0pt}}
```

Save the path as `\braid@this@path`.

```
\pgfsyssoftpath@getcurrentpath{\braid@this@path}
\else
```

We have seen this before, so we simply copy the associated path in to `\braid@this@path`.

```
\expandafter\let\expandafter\braid@this@path%
\csname braid@strand@\braid@this@path@origin\endcsname
\fi
```

Now we do the same again with the other strand in the crossing.

```
\expandafter\let\expandafter\braid@next@path@origin%
\csname braid@strand@\braid@next@strand @origin\endcsname
\ifx\braid@next@path@origin\relax
\let\braid@next@path@origin\braid@next@strand
```

```
  \pgfsyssoftpath@setcurrentpath{\@empty}
  \pgfpathmoveto{\pgfpoint{\braid@nx}{0pt}}
  \pgfsyssoftpath@getcurrentpath{\braid@next@path}
  \else
  \expandafter\let\expandafter\braid@next@path%
  \csname braid@strand@\braid@next@path@origin\endcsname
  \fi
```

Now that we have the paths for our two strands, we extend them to the next level. We start by selecting the first path.

```
  \pgfsyssoftpath@setcurrentpath{\braid@this@path}
```

Draw a line down to the current level, note that this line is always non-trivial since we shifted the corners of the crossing in a little.

```
  \pgfpathlineto{\pgfqpoint{\braid@tx}{\braid@ty}}
```

Curve across to the next position. Depending on the crossing type, we either have a single curve or we have to break it in two. Our gap is to interrupt at times determined by the gap key.

```
\pgfmathsetmacro{\braid@gst}{0.5 - \pgfkeysvalueof{/pgf/braid/gap}}%
\pgfmathsetmacro{\braid@gend}{0.5 + \pgfkeysvalueof{/pgf/braid/gap}}%
\ifx\braid@crossing@type\braid@over@cross
```

We're on the overpass, so just one curve needed.

```
  \pgfpathcurveto{\pgfqpoint{\braid@tx}{\braid@cy}}%
  {\pgfqpoint{\braid@nx}{\braid@dy}}%
  {\pgfqpoint{\braid@nx}{\braid@ny}}
\else
```

We're on the underpass, so we need to interrupt our path to allow the other curve to go past.

```
  \pgfpathcurvebetweentimecontinue{0}{\braid@gst}%
  {\pgfqpoint{\braid@tx}{\braid@ty}}%
  {\pgfqpoint{\braid@tx}{\braid@cy}}%
  {\pgfqpoint{\braid@nx}{\braid@dy}}%
  {\pgfqpoint{\braid@nx}{\braid@ny}}%
  \pgfpathcurvebetweentime{\braid@gend}{1}%
  {\pgfqpoint{\braid@tx}{\braid@ty}}%
  {\pgfqpoint{\braid@tx}{\braid@cy}}%
  {\pgfqpoint{\braid@nx}{\braid@dy}}%
  {\pgfqpoint{\braid@nx}{\braid@ny}}
\fi
```

We're done with this path, so now we save it.

```
  \pgfsyssoftpath@getcurrentpath{\braid@this@path}
```

Now do the same with the second path.

```
  \pgfsyssoftpath@setcurrentpath{\braid@next@path}
  \pgfpathlineto{\pgfqpoint{\braid@nx}{\braid@ty}}
\ifx\braid@crossing@type\braid@over@cross
  \pgfpathcurvebetweentimecontinue{0}{\braid@gst}%
  {\pgfqpoint{\braid@nx}{\braid@ty}}%
  {\pgfqpoint{\braid@nx}{\braid@cy}}%
  {\pgfqpoint{\braid@tx}{\braid@dy}}%
  {\pgfqpoint{\braid@tx}{\braid@ny}}
  \pgfpathcurvebetweentime{\braid@gend}{1}%
```

```
  {\pgfqpoint{\braid@nx}{\braid@ty}}%
  {\pgfqpoint{\braid@nx}{\braid@cy}}%
  {\pgfqpoint{\braid@tx}{\braid@dy}}%
  {\pgfqpoint{\braid@tx}{\braid@ny}}
\else
  \pgfpathcurveto{\pgfqpoint{\braid@nx}{\braid@cy}}%
  {\pgfqpoint{\braid@tx}{\braid@dy}}%
  {\pgfqpoint{\braid@tx}{\braid@ny}}
\fi
  \pgfsyssoftpath@getcurrentpath{\braid@next@path}
```

Now save the paths to their proper macros again.

```
  \expandafter\let%
  \csname braid@strand@\braid@this@path@origin \endcsname%
  \braid@this@path
  \expandafter\let%
  \csname braid@strand@\braid@next@path@origin \endcsname%
  \braid@next@path
```

Now update the origins

```
  \expandafter\let%
  \csname braid@strand@\braid@this@strand @origin\endcsname%
  \braid@next@path@origin
  \expandafter\let%
  \csname braid@strand@\braid@next@strand @origin\endcsname%
  \braid@this@path@origin
```

increment the strand counter, if necessary

```
  \pgfmathparse{\value{braid@strands} < \braid@next@strand ?
    "\noexpand\setcounter{braid@strands}{\braid@next@strand}" : ""}
  \pgfmathresult
```

And merrily go on our way with the next bit of the braid specification.

```
  \braid@process%
}
```

(*End definition for* `\braid@add@crossing`.)

`\braid@olabel@strand` This macro allows us to label a strand just before a crossing. The first argument is the strand number at that particular crossing and the second is the label. We also save the current height. This version takes the strand number as meaning the *original* ordering.

```
\newcommand{\braid@olabel@strand}[3][]{%
  \edef\braid@tmp{{\the\value{braid@level}}}%
  \expandafter\ifx\csname braid@strand@#2@origin\endcsname\relax
  \g@addto@macro\braid@tmp{{#2}}%
  \else
  \edef\braid@tmpa{{\csname braid@strand@#2@origin\endcsname}}%
  \ge@addto@macro\braid@tmp{\braid@tmpa}%
  \fi
  \g@addto@macro\braid@tmp{{#3}{#1}}%
  \ge@addto@macro{\braid@strand@labels}{\braid@tmp}%
  \braid@process%
}
```

(*End definition for* `\braid@olabel@strand`.)

\braid@clabel@strand This macro allows us to label a strand just before a crossing. The first argument is the strand number at that particular crossing and the second is the label. We also save the current height. This version takes the strand number as meaning the *current* ordering.

```
\newcommand{\braid@clabel@strand}[3][]{%
  \edef\braid@tmp{{\the\value{braid@level}}}%
  \g@addto@macro\braid@tmp{{#2}{#3}{#1}}%
  \ge@addto@macro{\braid@strand@labels}{\braid@tmp}%
  \braid@process%
}
```

(*End definition for* `\braid@clabel@strand`.)

\braid@floors@trim The list of floors, if given, will start with a superfluous comma. This removes it.

```
\def\braid@floors@trim,{}
```

(*End definition for* `\braid@floors@trim`.)

\braid@render@floor This is the default rendering for floors: it draws a rectangle.

```
\def\braid@render@floor{%
    \draw (\floorsx,\floorsy) rectangle (\floorex,\floorey);
}
```

(*End definition for* `\braid@render@floor`.)

\braid@render@strand@labels This starts rendering the labels on the strands at the crossings.

```
\def\braid@render@strand@labels#1{%
  \def\braid@tmp{#1}%
  \ifx\braid@tmp\pgfutil@empty
  \let\braid@next=\pgfutil@gobble
  \else
  \let\braid@next=\braid@@render@strand@labels
  \fi
  \braid@next{#1}%
}
```

(*End definition for* `\braid@render@strand@labels`.)

\braid@@render@strand@labels This is the actual renderer.

```
\def\braid@@render@strand@labels#1#2#3#4{%
  \begingroup
  \pgfscope
  \let\tikz@options=\pgfutil@empty
  \let\tikz@mode=\pgfutil@empty
  \let\tik@transform=\pgfutil@empty
  \let\tikz@fig@name=\pgfutil@empty
  \tikzset{/pgf/braid/strand label,#4}%
  \braid@nx=#2\braid@width
  \braid@ny=#1\braid@height
  \advance\braid@ny by \braid@eh
  \advance\braid@ny by \braid@height
  \pgftransformshift{\pgfqpoint{\braid@nx}{\braid@ny}}%
  \tikz@options
  \setbox\pgfnodeparttextbox=\hbox%
  \bgroup%
```

```
\tikzset{every text node part/.try}%
\ifx\tikz@textopacity\pgfutil@empty%
\else%
\pgfsetfillopacity{\tikz@textopacity}%
\pgfsetstrokeopacity{\tikz@textopacity}%
\fi%
\pgfinterruptpicture%
\tikz@textfont%
\ifx\tikz@text@width\pgfutil@empty%
\else%
\begingroup%
\pgfmathsetlength{\pgf@x}{\tikz@text@width}%
\pgfutil@minipage[t]{\pgf@x}\leavevmode\hbox{}%
\tikz@text@action%
\fi%
\tikz@atbegin@node%
\bgroup%
\aftergroup\unskip%
\ifx\tikz@textcolor\pgfutil@empty%
\else%
\pgfutil@colorlet{.}{\tikz@textcolor}%
\fi%
\pgfsetcolor{.}%
\setbox\tikz@figbox=\box\pgfutil@voidb@x%
\tikz@uninstallcommands%
\tikz@halign@check%
\ignorespaces%
#3
\egroup
\tikz@atend@node%
\ifx\tikz@text@width\pgfutil@empty%
\else%
\pgfutil@endminipage%
\endgroup%
\fi%
\endpgfinterruptpicture%
\egroup%
 \ifx\tikz@text@width\pgfutil@empty%
  \else%
    \pgfmathsetlength{\pgf@x}{\tikz@text@width}%
    \wd\pgfnodeparttextbox=\pgf@x%
  \fi%
  \ifx\tikz@text@height\pgfutil@empty%
  \else%
    \pgfmathsetlength{\pgf@x}{\tikz@text@height}%
    \ht\pgfnodeparttextbox=\pgf@x%
  \fi%
  \ifx\tikz@text@depth\pgfutil@empty%
  \else%
    \pgfmathsetlength{\pgf@x}{\tikz@text@depth}%
    \dp\pgfnodeparttextbox=\pgf@x%
  \fi%
\pgfmultipartnode{\tikz@shape}{\tikz@anchor}{\tikz@fig@name}{%
  {\begingroup\tikz@finish}%
```

```
  }%
  \endpgfscope
  \endgroup
  \braid@render@strand@labels%
}
```

(*End definition for* `\braid@@render@strand@labels`.)

`\braid@render` This is called at the end of the braid and it renders the braids and floors according to whatever has been built up up to now.

```
\def\braid@render{
```

Check for floors since we do them first.

```
  \ifx\braid@floors\@empty
  \else
```

Have some floors, start a scope and prepare to render them.

```
    \pgfsys@beginscope
```

Clear the path (just to be sure).

```
    \pgfsyssoftpath@setcurrentpath{\empty}
```

Trim the initial comma off the list of floors.

```
    \edef\braid@floors{\expandafter\braid@floors@trim\braid@floors}
```

Initialise our horizontal coordinates.

```
    \braid@tx=\braid@width
    \advance\braid@tx by \braid@eh
    \braid@nx=\value{braid@strands}\braid@width
    \advance\braid@nx by -\braid@eh
```

Loop over the list of floors.

```
    \foreach \braid@f in \braid@floors {
      \pgfsys@beginscope
```

Figure out the vertical coordinates for the current floor.

```
      \braid@ty=\braid@f\braid@height
      \advance\braid@ty by \braid@eh
      \advance\braid@ty by \braid@height
      \braid@ny=\braid@ty
      \advance\braid@ny by \braid@height
```

Save the coordinates for use in the floor rendering macro.

```
      \edef\floorsx{\the\braid@tx}
      \edef\floorsy{\the\braid@ty}
      \edef\floorex{\the\braid@nx}
      \edef\floorey{\the\braid@ny}
      \let\tikz@options=\pgfutil@empty
```

Load general floor style options.

```
    \expandafter\tikzset\expandafter{\braid@floors@style}
```

Load any style options specific to this floor. We're actually offset by 2 from what the user thinks the floor level is.

```
      \pgfmathtruncatemacro{\braid@ff}{\braid@f+2}
```

Load the relevant floor style, if it exists.

```
    \expandafter\let\expandafter\braid@floor@style%
    \csname braid@options@floor@\braid@ff\endcsname
    \ifx\braid@floor@style\relax
    \else
```

There is a floor style for this level, so process it.

```
    \expandafter\tikzset\expandafter{\braid@floor@style}%
    \fi
```

The `\tikzset` just parses the options, we need to call `\tikz@options` to actually set them.

```
\tikz@options
```

Now we call the rendering code.

```
\braid@render@floor
```

Done! End the scope for *this* floor and go again.

```
\pgfsys@endscope
    }
```

Done rendering floors, end the scope.

```
    \pgfsys@endscope
    \fi
```

Finished with floors (if we had them), now get on with the strands.

```
  \stepcounter{braid@level}
  \foreach \braid@k in {1,...,\value{braid@strands}} {
```

Start a local scope to ensure we don't mess with other braids

```
    \pgfsys@beginscope
```

Default is to draw each braid

```
    \tikz@mode@drawtrue%
    \let\tikz@mode=\pgfutil@empty
    \let\tikz@options=\pgfutil@empty
```

(x,y) coordinates of bottom of strand

```
    \braid@tx=\braid@k\braid@width
    \braid@ty=\value{braid@level}\braid@height
    \advance\braid@ty by 2\braid@eh
```

Try to find the starting point of this strand

```
    \expandafter\let\expandafter\braid@path@origin%
    \csname braid@strand@\braid@k @origin\endcsname
    \ifx\braid@path@origin\relax
```

If that doesn't exist, we'll just draw a straight line so we move to the top of the current position

```
    \pgfsyssoftpath@setcurrentpath{\@empty}
    \pgfpathmoveto{\pgfqpoint{\braid@tx}{0pt}}
    \let\braid@path@origin\braid@k
    \else
```

If the path does exist, we load it

```
    \expandafter\let\expandafter\braid@path%
    \csname braid@strand@\braid@path@origin\endcsname
    \pgfsyssoftpath@setcurrentpath{\braid@path}
    \fi
```

Extend the path to the bottom

```
    \pgflineto{\pgfqpoint{\braid@tx}{\braid@ty}}
```

Load common style options

```
    \expandafter\tikzset\expandafter{\braid@style}
```

Load any style options specific to this strand

```
    \expandafter\let\expandafter\braid@style%
    \csname braid@options@strand@\braid@path@origin\endcsname
    \ifx\braid@style\relax
    \else
    \expandafter\tikzset\expandafter{\braid@style}
    \fi
\braid@options
    \tikz@mode
    \tikz@options
```

This is the command that actually draws the strand.

```
      \edef\tikz@temp{\noexpand\pgfusepath{%
          \iftikz@mode@draw draw\fi%
      }}%
      \tikz@temp
```

If our braid has a name, we label the ends of the strand.

```
\ifx\braid@name\pgfutil@empty
\else
```

Label the ends of the strand.

```
\coordinate (\braid@name-\braid@path@origin-e) at (\braid@tx,\braid@ty);
\coordinate (\braid@name-rev-\braid@k-e) at (\braid@tx,\braid@ty);
\braid@nx=\braid@path@origin\braid@width
\coordinate (\braid@name-\braid@path@origin-s) at (\braid@nx,0pt);
\coordinate (\braid@name-rev-\braid@k-s) at (\braid@nx,0pt);
\fi
```

Done with this strand, close the scope and do the next one.

```
   \pgfsys@endscope
  }
```

If our braid has a name, we also want to label the centre.

```
    \ifx\braid@name\pgfutil@empty
    \else
    \braid@tx=\value{braid@strands}\braid@width
    \braid@ty=\value{braid@level}\braid@height
    \advance\braid@ty by 2\braid@eh
    \advance\braid@tx by \braid@width
    \braid@tx=.5\braid@tx
    \braid@ty=.5\braid@ty
    \coordinate (\braid@name) at (\braid@tx,\braid@ty);
    \fi
```

Now we label the strands if needed.

```
  \ifx\braid@strand@labels\pgfutil@empty
  \else
   \expandafter\braid@render@strand@labels\braid@strand@labels{}%
  \fi
```

All done now, close the scope and end the group (which was opened right at the start).

```
    \pgfsys@endscope
  \endgroup}
```

(*End definition for* `\braid@render`.)

`\braid@start` This starts off the braid, initialising a load of stuff. We start a PGF scope, set the level to −1, the label, floors, and name to empty, process any options we're given, and save certain lengths for later use..

```
\def\braid@start#1{%
  \pgfsys@beginscope
  \setcounter{braid@level}{-1}%
  \let\braid@label\@empty
  \let\braid@strand@labels\@empty
  \let\braid@floors\@empty
  \let\braid@name\empty
  \let\clabel=\braid@clabel@strand
  \let\olabel=\braid@olabel@strand
  \pgfkeys{/pgf/braid/.cd,#1}%
  \ifbraid@strand@labels@origin
  \let\label=\braid@olabel@strand
  \else
  \let\label=\braid@clabel@strand
  \fi
  \let\braid@options\tikz@options
  \tikz@transform
  \setcounter{braid@strands}{%
    \pgfkeysvalueof{/pgf/braid/number of strands}}%
  \braid@width=\pgfkeysvalueof{/pgf/braid/width}%
  \braid@height=\pgfkeysvalueof{/pgf/braid/height}%
  \braid@eh=\pgfkeysvalueof{/pgf/braid/border height}%
  \pgfkeysgetvalue{/pgf/braid/control factor}{\braid@cf}%
  \pgfkeysgetvalue{/pgf/braid/nudge factor}{\braid@nf}%
  \braid@height=-\braid@height
  \braid@eh=-\braid@eh
  \braid@increase@leveltrue
  \braid@process@start
}
```

(*End definition for* `\braid@start`.)

These are the lengths we'll use as we construct the braid

```
\newdimen\braid@width
\newdimen\braid@height
\newdimen\braid@tx
\newdimen\braid@ty
\newdimen\braid@nx
\newdimen\braid@ny
\newdimen\braid@cy
\newdimen\braid@dy
\newdimen\braid@eh
```

An if to decide whether or not to step to the next level or not

```
\newif\ifbraid@increase@level
```

An if to decide whether label indices should be absolute or not

```
\newif\ifbraid@strand@labels@origin
```

Some initial values

```
\let\braid@style\pgfutil@empty
\let\braid@floors@style\pgfutil@empty
\def\braid@over@cross{1}
```

Counters to track the strands and the levels.

```
\newcounter{braid@level}
\newcounter{braid@strands}
```

All the keys we'll use.

```
\pgfkeys{
```

Handle unknown keys by passing them to `pgf` and `tikz`.

```
    /tikz/braid/.search also={/pgf},
    /pgf/braid/.search also={/pgf,/tikz},
```

Our "namespace" is `/pgf/braid`.

```
    /pgf/braid/.cd,
    number of strands/.initial=0,
    height/.initial=1cm,
    width/.initial=1cm,
    gap/.initial=.1,
    border height/.initial=.25cm,
    control factor/.initial=.5,
    nudge factor/.initial=.05,
    name/.code={%
      \def\braid@name{#1}%
    },
    at/.code={%
      \braid@relocate{#1}%
    },
    floor command/.code={%
      \def\braid@render@floor{#1}%
    },
    style strands/.code 2 args={%
      \def\braid@temp{#2}%
      \braidset{style each strand/.list={#1}}%
    },
    style each strand/.code={%
      \expandafter\edef%
      \csname braid@options@strand@#1\endcsname{\braid@temp}%
    },
    style floors/.code 2 args={%
      \def\braid@temp{#2}%
      \braidset{style each floor/.list={#1}}%
    },
    style each floor/.code={%
      \expandafter\edef%
      \csname braid@options@floor@#1\endcsname{\braid@temp}%
    },
    style all floors/.code={%
      \def\braid@floors@style{#1}
    },
```

```
    strand label/.style={},
    strand label by origin/.is if=braid@strand@labels@origin,
}
```

`\braidset` Shorthand for setting braid-specific keys.

```
\def\braidset#1{%
  \pgfkeys{/pgf/braid/.cd,#1}}
```

(*End definition for* `\braidset`. *This function is documented on page* **??**.)

```
⟨*library⟩
⟨@@=braid⟩
```

4 Reimplementation as a TikZ Library

Life is so much easier with LaTeX3.

```
\ProvidesFile{tikzlibrarybraids.code.tex}[%
  2019/03/20 v2.0 Tikz/PGF library for drawing braid diagrams%
]
\RequirePackage{expl3}
\ExplSyntaxOn
```

Define all the variables we'll be using.

```
\tl_new:N \l__braid_tmpa_tl
\tl_new:N \l__braid_tmpb_tl
\tl_new:N \l__braid_tmpc_tl
\tl_new:N \l__braid_tmpd_tl
\tl_new:N \l__braid_anchor_strand_tl
\tl_new:N \l__braid_anchor_level_tl
\fp_new:N \l__braid_height_fp
\fp_new:N \l__braid_width_fp
\fp_new:N \l__braid_nudge_fp
\fp_new:N \l__braid_control_fp
\fp_new:N \l__braid_ctrlax_fp
\fp_new:N \l__braid_ctrlay_fp
\fp_new:N \l__braid_ctrlbx_fp
\fp_new:N \l__braid_ctrlby_fp
\fp_new:N \l__braid_endx_fp
\fp_new:N \l__braid_endy_fp
\fp_new:N \l__braid_anchor_x_fp
\fp_new:N \l__braid_anchor_y_fp
\int_new:N \l__braid_tmpa_int
\int_new:N \l__braid_length_int
\int_new:N \l__braid_strands_int
\int_new:N \l__braid_crossing_int
\int_new:N \l__braid_anchor_level_int
\int_new:N \l__braid_floor_int
\seq_new:N \l__braid_word_seq
\seq_new:N \l__braid_crossing_seq
\seq_new:N \l__braid_anchor_seq
\seq_new:N \l__braid_floors_seq
\str_new:N \l__braid_tmpa_str
\str_new:N \l__braid_sup_str
\str_set:Nn \l__braid_sup_str {^}
```

```
\str_new:N \l__braid_sub_str
\str_set:Nn \l__braid_sub_str {_}
\str_new:N \l__braid_hyphen_str
\str_set:Nn \l__braid_hyphen_str {-}
\str_new:N \l__braid_bar_str
\str_set:Nn \l__braid_bar_str {|}
\str_new:N \l__braid_one_str
\str_set:Nn \l__braid_one_str {1}
\bool_new:N \l__braid_step_level_bool
\bool_new:N \l__braid_swap_crossing_bool
\bool_new:N \l__braid_floor_bool
\prop_new:N \l__braid_strands_prop
\prop_new:N \l__braid_permutation_prop
\prop_new:N \l__braid_crossing_permutation_prop
\prop_new:N \l__braid_inverse_prop
\prop_new:N \l__braid_anchor_prop
```

Our interface is through a TikZ pic.

```
\tikzset{
  braid/.pic={
    code={
      \__braid_parse_word:n {#1}
      \__braid_count:
      \__braid_render:
    }
  },
  floor/.pic={
    code={
      \path[pic~ actions, draw=none] (0,0) rectangle (1,1);
      \path[pic~ actions, fill=none] (0,0) -- (1,0) (0,1) -- (1,1);
    }
  },
  /tikz/braid/.search~ also={/tikz},
  braid/.cd,
```

The various TikZ parameters for the braid.

The anchor determines which part of the braid is located at the position specified by the pic. It can be of the form `n-m` where `n` is a strand number and +m+ is a crossing level. The strand number can be either a number or `rev-n` to use the ending numbering of the strands. The crossing level can also be `s` or `e` which means the actual start or end of the strand (including the border).

```
    anchor/.initial=1-s,
```

`number of strands` sets a minimum for the number of strands in the braid (otherwise, it is set by the strands used in the specified crossings).

```
    number~ of~ strands/.initial=0,
```

`height` is the distance between crossings (can be negative).

```
    height/.initial=-1cm,
```

`width` is the distance between strands (can be negative).

```
    width/.initial=1cm,
```

`gap` is for determining the gap in the under-strand of a crossing.

```
    gap/.initial=.05,
```

`border height` is a length added at the start and end of each strand.

```
    border~ height/.initial=.25cm,
```

`floor border` is added to the width of any floors

```
    floor~ border/.initial=.25cm,
```

`floors` is a list of floors to draw, specified as a cslist of coordinates as (x,y,w,h,a) in which the units are numbers of strands and crossing levels. The parameters are: coordinates of lower left corner, width, height, (optional) name for styling.

```
    add~ floor/.code={
      \seq_push:Nn \l__braid_floors_seq {#1}
    },
```

`control factor` determines the proportion of the `height` used for the control points.

```
    control~ factor/.initial=.5,
```

`nudge factor` is used to compress each crossing slightly within its rectangle.

```
    nudge~ factor/.initial=.05
}
```

`\__braid_parse_word:Nn` Parse the braid word as a token list and convert it into a sequence.

```
\cs_new_nopar:Npn \__braid_parse_word:n #1
{
  \seq_clear:N \l__braid_word_seq
  \tl_clear:N \l__braid_tmpa_tl
  \tl_set:Nn \l__braid_tmpb_tl {#1}

  \bool_until_do:nn { \tl_if_empty_p:N \l__braid_tmpb_tl }
  {
```

We step through the braid specification, looking for special characters. To avoid catcode issues, the comparison is as strings. Some actions may involve consuming more tokens from the list so we can't do a simple `map_inline` but have to keep stripping off the head token.

The idea is to store information about the current crossing in a token list (noting that it may be specified in a variety of orders) and then when we're sure we have all the information we add it to our sequence of crossings.

```
    \str_set:Nx \l__braid_tmpa_str {\tl_head:N \l__braid_tmpb_tl}
    \tl_set:Nx \l__braid_tmpb_tl {\tl_tail:N \l__braid_tmpb_tl}
    \str_case_e:nnTF {\l__braid_tmpa_str}
    {
```

Underscore introduces the crossing numbers

```
      {_}
     {
       \tl_put_right:Nx \l__braid_tmpa_tl
       {
         \exp_not:N \__braid_parse_index:n {\tl_head:N \l__braid_tmpb_tl}
       }
       \tl_set:Nx \l__braid_tmpb_tl {\tl_tail:N \l__braid_tmpb_tl}
     }
```

Power is used to indicate inverse.

```
      {^}
      {
        \tl_put_left:Nx \l__braid_tmpa_tl
        {
          \exp_not:N \__braid_parse_exponent:n {\tl_head:N \l__braid_tmpb_tl}
        }
        \tl_set:Nx \l__braid_tmpb_tl {\tl_tail:N \l__braid_tmpb_tl}
      }
```

Bar is for floors.

```
      {|}
      {
       \tl_if_empty:NF \l__braid_tmpa_tl
       {
         \seq_put_right:NV \l__braid_word_seq \l__braid_tmpa_tl
         \tl_clear:N \l__braid_tmpa_tl
       }

       \tl_set:Nn \l__braid_tmpa_tl {
         \bool_set_false:N \l__braid_step_level_bool
         \bool_set_true:N \l__braid_floor_bool
       }
       \seq_put_right:NV \l__braid_word_seq \l__braid_tmpa_tl
       \tl_clear:N \l__braid_tmpa_tl
      }
```

Hyphen says the next crossing is on the same level as the current one.

```
      {-}
      {
        \tl_put_right:Nn \l__braid_tmpa_tl
        {
          \bool_set_false:N \l__braid_step_level_bool
        }
      }
```

1 is for the identity (i.e., no crossing but still have a level). We put a nop token on the list so that it is no longer empty.

```
      {1}
      {
       \tl_if_empty:NF \l__braid_tmpa_tl
       {
         \seq_put_right:NV \l__braid_word_seq \l__braid_tmpa_tl
         \tl_clear:N \l__braid_tmpa_tl
       }
        \tl_put_right:Nn \l__braid_tmpa_tl {\__braid_do_identity:}
      }
```

Ignore spaces.

```
      {~}
      {
      }
     }
     {
     }
     {
```

If we get an unrecognised token, it's our trigger to start accumulating information for the next crossing.

```
      \tl_if_empty:NF \l__braid_tmpa_tl
      {
        \seq_put_right:NV \l__braid_word_seq \l__braid_tmpa_tl
        \tl_clear:N \l__braid_tmpa_tl
      }
    }
  }
```

At the end, we also put our current token list on the word sequence.

```
  \tl_if_empty:NF \l__braid_tmpa_tl
  {
    \seq_put_right:NV \l__braid_word_seq \l__braid_tmpa_tl
    \tl_clear:N \l__braid_tmpa_tl
  }
}
```

(*End definition for* `\__braid_parse_word:Nn`.)

`\__braid_parse_index:n` Parse an index, saving it in a sequence with the two indices such that the first goes over the second.

```
\cs_new_nopar:Npn \__braid_parse_index:n #1
{
  \seq_set_from_clist:Nn \l__braid_crossing_seq {#1}
  \int_compare:nT {\seq_count:N \l__braid_crossing_seq == 1}
  {
    \seq_put_right:Nx \l__braid_crossing_seq {\int_eval:n {#1 + 1} }
  }
  \bool_if:NT \l__braid_swap_crossing_bool
  {
    \seq_pop_left:NN \l__braid_crossing_seq \l__braid_tmpa_tl
    \seq_put_right:NV \l__braid_crossing_seq \l__braid_tmpa_tl
  }
}
```

(*End definition for* `\__braid_parse_index:n`.)

`\__braid_parse_exponent:n` Parse an exponent, basically testing to see if it is -1 in which case our crossing numbers should be reversed..

```
\cs_new_nopar:Npn \__braid_parse_exponent:n #1
{
  \int_compare:nTF {#1 == -1}
  {
    \bool_set_true:N \l__braid_swap_crossing_bool
  }
  {
    \bool_set_false:N \l__braid_swap_crossing_bool
  }
}
```

(*End definition for* `\__braid_parse_exponent:n`.)

__braid_do_identity:

```
\cs_new_nopar:Npn \__braid_do_identity:
{
}
```

(*End definition for* `\__braid_do_identity:`.)

__braid_count:NNN Work out how big the braid is by counting strands and levels. We also figure out the permutation from the start to end of the strands. This is useful for labelling various parts of the braid.

```
\cs_new_nopar:Npn \__braid_count:
{
  \int_zero:N \l__braid_length_int
  \int_set:Nn \l__braid_strands_int {\__braid_value:n {number~of~strands}}
  \prop_clear:N \l__braid_permutation_prop
  \prop_clear:N \l__braid_crossing_permutation_prop
  \prop_clear:N \l__braid_anchor_prop
  \prop_clear:N \l__braid_inverse_prop

  \seq_map_inline:Nn \l__braid_word_seq
  {
```

Clear the crossing sequence and assume we're going to step the level.

```
    \seq_clear:N \l__braid_crossing_seq
    \bool_set_true:N \l__braid_step_level_bool
    \bool_set_false:N \l__braid_swap_crossing_bool
```

Run the details of this crossing.

```
    ##1
```

If we're increasing the level (no hyphen), do so.

```
    \bool_if:NT \l__braid_step_level_bool
    {
      \int_incr:N \l__braid_length_int
    }
```

If we have a crossing, check we have enough strands to cover it.

```
    \seq_if_empty:NF \l__braid_crossing_seq
    {
      \int_set:Nn \l__braid_strands_int
      {
        \int_max:nn
        {
          \int_max:nn {\l__braid_strands_int}
          { \seq_item:Nn \l__braid_crossing_seq {1}}
        }
        {
          \seq_item:Nn \l__braid_crossing_seq {2}
        }
      }
    }
  }
```

Now that we know how many strands we have, we can initialise our permutation props. One will hold the overall permutation, the other will keep track of our current permutation.

```
\int_step_inline:nnnn {1} {1} {\l__braid_strands_int}
{
  \prop_put:Nnn \l__braid_permutation_prop {##1} {##1}
  \prop_put:Nnn \l__braid_anchor_prop {##1} {##1}
  \prop_put:Nnn \l__braid_crossing_permutation_prop {##1} {##1}
}
```

Now we step through the braid word again and record the permutations so that we can calculate the overall permutation defined by the braid.

We will also figure out our shift from the anchor, so first we need to get some information about the anchor.

```
\tl_set:Nx \l__braid_tmpa_tl {\__braid_value:n {anchor}}
\seq_set_split:NnV \l__braid_anchor_seq {-} \l__braid_tmpa_tl

\tl_set:Nx \l__braid_tmpa_tl {\seq_item:Nn \l__braid_anchor_seq {1}}
\tl_if_eq:VnTF \l__braid_tmpa_tl {rev}
{
  \tl_set:Nx \l__braid_anchor_strand_tl {\seq_item:Nn \l__braid_anchor_seq {2}}
  \tl_set:Nx \l__braid_anchor_level_tl {\seq_item:Nn \l__braid_anchor_seq {3}}
}
{
  \tl_set:Nx \l__braid_anchor_strand_tl {\seq_item:Nn \l__braid_anchor_seq {1}}
  \tl_set:Nx \l__braid_anchor_level_tl {\seq_item:Nn \l__braid_anchor_seq {2}}
}
```

The important information is as to the level at which the requested anchor resides. If it is at the end or start of a strand, we set the level to -1 so that it never matches a level number.

```
\tl_if_eq:VnTF \l__braid_anchor_level_tl {s}
{
  \int_set:Nn \l__braid_anchor_level_int {-1}
}
{
  \tl_if_eq:VnTF \l__braid_anchor_level_tl {e}
  {
    \int_set:Nn \l__braid_anchor_level_int {-1}
  }
  {
    \int_set:Nn \l__braid_anchor_level_int
    {\tl_use:N \l__braid_anchor_level_tl}
  }
}

\int_zero:N \l__braid_crossing_int
\int_incr:N \l__braid_crossing_int
\seq_map_inline:Nn \l__braid_word_seq
{
  \bool_set_true:N \l__braid_step_level_bool
  \seq_clear:N \l__braid_crossing_seq
  \bool_set_false:N \l__braid_swap_crossing_bool
  ##1
```

```
    \seq_if_empty:NF \l__braid_crossing_seq
    {
      \prop_get:NxN \l__braid_permutation_prop
      {
        \seq_item:Nn \l__braid_crossing_seq {1}
      } \l__braid_tmpa_tl
      \prop_get:NxN \l__braid_permutation_prop
      {
        \seq_item:Nn \l__braid_crossing_seq {2}
      } \l__braid_tmpb_tl

      \prop_put:NxV \l__braid_permutation_prop
      {
        \seq_item:Nn \l__braid_crossing_seq {2}
      } \l__braid_tmpa_tl
      \prop_put:NxV \l__braid_permutation_prop
      {
        \seq_item:Nn \l__braid_crossing_seq {1}
      } \l__braid_tmpb_tl
    }
```

See if the current level is what was requested by the anchor.

```
    \int_compare:nT {\l__braid_crossing_int = \l__braid_anchor_level_int}
    {
      \prop_set_eq:NN \l__braid_anchor_prop \l__braid_permutation_prop
    }
    \bool_if:NT \l__braid_step_level_bool
    {
      \int_incr:N \l__braid_crossing_int
    }
  }
```

This inverts the anchor permutation.

```
  \int_step_inline:nnnn {1} {1} {\l__braid_strands_int}
  {
    \prop_get:NnN \l__braid_anchor_prop {##1} \l__braid_tmpa_tl
    \prop_put:NVn \l__braid_inverse_prop \l__braid_tmpa_tl {##1}
  }
  \prop_set_eq:NN \l__braid_anchor_prop \l__braid_inverse_prop
```

This inverts the full permutation.

```
  \int_step_inline:nnnn {1} {1} {\l__braid_strands_int}
  {
    \prop_get:NnN \l__braid_permutation_prop {##1} \l__braid_tmpa_tl
    \prop_put:NVn \l__braid_inverse_prop \l__braid_tmpa_tl {##1}
  }
```

Now that we have the inverse, we can figure out our anchor. First, see if we requested a strand by its position at the end of the braid.

```
  \tl_set:Nx \l__braid_tmpa_tl {\seq_item:Nn \l__braid_anchor_seq {1}}
  \tl_if_eq:VnT \l__braid_tmpa_tl {rev}
  {
    \prop_get:NVN \l__braid_permutation_prop
    \l__braid_anchor_strand_tl \l__braid_anchor_strand_tl
  }
```

```
  \tl_if_eq:VnF \l__braid_anchor_level_tl {s}
  {
    \tl_if_eq:VnTF \l__braid_anchor_level_tl {e}
    {
      \prop_get:NVN \l__braid_inverse_prop
      \l__braid_anchor_strand_tl \l__braid_anchor_strand_tl
    }
    {
      \prop_get:NVN \l__braid_anchor_prop
      \l__braid_anchor_strand_tl \l__braid_anchor_strand_tl
    }
  }
}
```

(*End definition for* `\__braid_count:NNN`.)

`\__braid_dim_value:n` `\__braid_value:n` Extract a length or a value from a PGF key.

```
\cs_new_nopar:Npn \__braid_dim_value:n #1
{
  \dim_to_fp:n {\pgfkeysvalueof{/tikz/braid/#1}}
}
\cs_new_nopar:Npn \__braid_value:n #1
{
  \pgfkeysvalueof{/tikz/braid/#1}
}
```

(*End definition for* `\__braid_dim_value:n` *and* `\__braid_value:n`.)

`\__braid_render:` This is the macro that converts the braid word into TikZ paths.

```
\cs_generate_variant:Nn \prop_get:NnN {NxN}
\cs_generate_variant:Nn \prop_put:Nnn {NxV}
\cs_generate_variant:Nn \tl_if_eq:nnTF {VnTF}
\cs_generate_variant:Nn \tl_if_eq:nnF {VnF}
\cs_generate_variant:Nn \tl_if_eq:nnT {VnT}

\cs_new_nopar:Npn \__braid_render:
{
  \fp_set:Nn \l__braid_anchor_x_fp { - 1 * (\tl_use:N \l__braid_anchor_strand_tl - 1) * \__b

  \tl_if_eq:VnTF \l__braid_anchor_level_tl {s}
  {
    \fp_set:Nn \l__braid_anchor_y_fp {0}
  }
  {
    \tl_if_eq:VnTF \l__braid_anchor_level_tl {e}
    {
      \fp_set:Nn \l__braid_anchor_y_fp {
        -1 * \l__braid_length_int * \__braid_dim_value:n {height}
        - sign(\__braid_dim_value:n {height})
        * 2 * \__braid_dim_value:n {border~ height}
      }
    }
    {
      \fp_set:Nn \l__braid_anchor_y_fp {
```

```
        -1 * \l__braid_anchor_level_tl * \__braid_dim_value:n {height}
        - sign(\__braid_dim_value:n {height})
        * \__braid_dim_value:n {border~ height}
      }
    }
  }

  \begin{scope}[
    shift={
      (\fp_to_decimal:N \l__braid_anchor_x_fp pt,
      \fp_to_decimal:N \l__braid_anchor_y_fp pt
      )
    }
  ]
```

Initialise a prop for the individual strands.

```
  \prop_clear:N \l__braid_strands_prop
```

Initialise some lengths.

```
  \fp_zero:N \l__braid_height_fp
  \fp_zero:N \l__braid_nudge_fp
  \fp_zero:N \l__braid_control_fp
```

This holds our current `height` of our strands.

```
  \fp_set:Nn \l__braid_height_fp
  {
    sign(\__braid_dim_value:n {height})
    * \__braid_dim_value:n {border~ height}
  }
```

This holds the total `width` of our strands.

```
  \fp_set:Nn \l__braid_width_fp
  {
    (\l__braid_strands_int - 1) * \__braid_dim_value:n {width}
    + 2 * sign(\__braid_dim_value:n{width})
    * \__braid_dim_value:n {floor~ border}
  }
```

Each crossing actually starts a little bit into the crossing space, as defined by the `nudge factor`.

```
  \fp_set:Nn \l__braid_nudge_fp
  {
    \__braid_value:n {nudge~ factor} * \__braid_dim_value:n {height}
  }
```

This sets where the control points for the crossing curves will be.

```
  \fp_set:Nn \l__braid_control_fp
  {
    \__braid_value:n {control~ factor} * \__braid_dim_value:n {height}
  }
  \fp_sub:Nn \l__braid_control_fp {\l__braid_nudge_fp}
```

Initialise our strand paths with a `\draw`.

```
  \int_step_inline:nnnn {1} {1} {\l__braid_strands_int}
  {
    \prop_get:NnN \l__braid_inverse_prop {##1} \l__braid_tmpa_tl
```

```
    \prop_put:Nnx \l__braid_strands_prop {##1}
    {
      \exp_not:N \draw[
        braid/every~ strand/.try,
        braid/strand~ ##1/.try
      ]
      \exp_not:N \__braid_moveto:nn {
        \fp_eval:n {(##1 - 1) * \__braid_dim_value:n {width} }
      } {0}
      \exp_not:N \__braid_lineto:nn  {
        \fp_eval:n {(##1 - 1) * \__braid_dim_value:n {width} }
      } { \fp_to_decimal:N \l__braid_height_fp}
    }
```

Add a load of coordinates at the start of each strand, indexed by both forward and backward strand numbers.

```
    \__braid_coordinate:xxxx {-##1-s} {-rev-\l__braid_tmpa_tl-s}
    {\fp_eval:n {(##1 - 1) * \__braid_dim_value:n {width} }} {0}

    \__braid_coordinate:xxxx {-##1-0} {-rev-\l__braid_tmpa_tl-0}
    {\fp_eval:n {(##1 - 1) * \__braid_dim_value:n {width} }}
    { \fp_to_decimal:N \l__braid_height_fp}
  }
```

Run through any extra floors requested.

```
  \seq_map_inline:Nn \l__braid_floors_seq
  {
    \tl_set:Nx \l__braid_tmpa_tl {\clist_item:nn {##1} {5}}
    \__braid_do_floor:Vxxxx \l__braid_tmpa_tl
    {\fp_eval:n
      {
        -1*sign(\__braid_dim_value:n{width})
        * \__braid_dim_value:n {floor~ border}
        + (\__braid_dim_value:n {width}) * (\clist_item:nn {##1} {1} - 1)
      }
      pt
    }
    {\fp_eval:n
      {
        \l__braid_height_fp + ( \__braid_dim_value:n {height} ) * (\clist_item:nn {##1} {2})
      }
      pt
    }
    {\fp_eval:n {
        ( (\clist_item:nn {##1} {3}) * \__braid_dim_value:n {width}
        + 2 * sign(\__braid_dim_value:n{width})
        * \__braid_dim_value:n {floor~ border} ) / \dim_to_fp:n {1cm}
      }
    }
    {\fp_eval:n {
        (\clist_item:nn {##1} {4}) * ( \__braid_dim_value:n {height} ) / \dim_to_fp:n {1cm}
      }
    }
  }
```

Keep track of the crossing level for the floor.

```
\int_zero:N \l__braid_crossing_int
\int_incr:N \l__braid_crossing_int

\seq_map_inline:Nn \l__braid_word_seq
{
```

Clear the flags for this segment of the braid word

```
  \seq_clear:N \l__braid_crossing_seq
  \bool_set_true:N \l__braid_step_level_bool
  \bool_set_false:N \l__braid_floor_bool
  \bool_set_false:N \l__braid_swap_crossing_bool
  ##1
```

If we're drawing a floor, do so straightaway.

```
  \bool_if:NT \l__braid_floor_bool
  {
    \__braid_do_floor:Vxxxx \l__braid_crossing_int
    {\fp_eval:n
      {
        -1*sign(\__braid_dim_value:n{width})
        * \__braid_dim_value:n {floor~ border}
      }
      pt
    }
    {\fp_to_decimal:N \l__braid_height_fp pt}
    {\fp_eval:n { \l__braid_width_fp / \dim_to_fp:n {1cm} }}
    {\fp_eval:n { ( \__braid_dim_value:n {height} ) / \dim_to_fp:n {1cm}}}
  }
```

If we have a crossing, process it.

```
  \seq_if_empty:NF \l__braid_crossing_seq
  {
```

Keep track of the current permutation.

```
    \prop_get:NxN \l__braid_crossing_permutation_prop
    {\seq_item:Nn \l__braid_crossing_seq {1}} \l__braid_tmpa_tl
    \prop_get:NxN \l__braid_crossing_permutation_prop
    {\seq_item:Nn \l__braid_crossing_seq {2}} \l__braid_tmpb_tl

    \prop_put:NxV \l__braid_crossing_permutation_prop
    {\seq_item:Nn \l__braid_crossing_seq {2}} \l__braid_tmpa_tl
    \prop_put:NxV \l__braid_crossing_permutation_prop
    {\seq_item:Nn \l__braid_crossing_seq {1}} \l__braid_tmpb_tl
```

Now get the strands corresponding to the ones involved in the crossing.

```
    \prop_get:NxN \l__braid_strands_prop
    {\seq_item:Nn \l__braid_crossing_seq {1}} \l__braid_tmpa_tl
    \prop_get:NxN \l__braid_strands_prop
    {\seq_item:Nn \l__braid_crossing_seq {2}} \l__braid_tmpb_tl
```

The over-strand is easy as that's a single curve.

```
    \tl_put_right:Nx \l__braid_tmpa_tl
    {
      \exp_not:N \__braid_lineto:nn

```

```
      {\fp_eval:n
        {
          (\seq_item:Nn \l__braid_crossing_seq {1} - 1)
          * \__braid_dim_value:n {width}
        }
      }
      {\fp_eval:n { \l__braid_height_fp + \l__braid_nudge_fp } }

      \exp_not:N \__braid_curveto:nnnnnn

      {0}
      {\fp_eval:n { \l__braid_control_fp}}

      {0}
      {\fp_eval:n {- \l__braid_control_fp}}

      {\fp_eval:n
        {
          (\seq_item:Nn \l__braid_crossing_seq {2} - 1)
          * \__braid_dim_value:n {width}
        }
      }
      {\fp_eval:n
        {
          \l__braid_height_fp
          + \__braid_dim_value:n {height}
          - \l__braid_nudge_fp
        }
      }
    }
```

The under-strand is a bit more complicated as we need to break it in the middle.

```
    \tl_put_right:Nx \l__braid_tmpb_tl
    {
      \exp_not:N \__braid_lineto:nn

      {\fp_eval:n
        {
          (\seq_item:Nn \l__braid_crossing_seq {2} - 1)
          * \__braid_dim_value:n {width}
        }
      }
      {\fp_eval:n { \l__braid_height_fp + \l__braid_nudge_fp } }

      \exp_not:N \__braid_curveto:nnnnnn

      {0}
      {
        \fp_eval:n {
          \l__braid_control_fp  * (.5 - \__braid_value:n {gap} )
        }
      }

      {
        \fp_eval:n {
```

```
      - (.5 - \__braid_value:n {gap} ) / 3 *
      \__braid_bezier_tangent:nnnnn
      {.5 - \__braid_value:n {gap} }
      {0}
      {0}
      {
        (\seq_item:Nn \l__braid_crossing_seq {1}
        - \seq_item:Nn \l__braid_crossing_seq {2})
        * \__braid_dim_value:n {width}
      }
      {
        (\seq_item:Nn \l__braid_crossing_seq {1}
        - \seq_item:Nn \l__braid_crossing_seq {2})
        * \__braid_dim_value:n {width}
      }
    }
  }
  {
    \fp_eval:n {
      -(.5 - \__braid_value:n {gap} ) / 3 *
      \__braid_bezier_tangent:nnnnn
      {.5 - \__braid_value:n {gap} }
      {0}
      {\l__braid_control_fp}
      {
        \__braid_dim_value:n {height}
        - 2* \l__braid_nudge_fp
        - \l__braid_control_fp
      }
      {\__braid_dim_value:n {height} - 2* \l__braid_nudge_fp}
    }
  }

  {
    \fp_eval:n {
      (\seq_item:Nn \l__braid_crossing_seq {2} - 1)
      * \__braid_dim_value:n {width} +
      \__braid_bezier_point:nnnnn
      {.5 - \__braid_value:n {gap} }
      {0}
      {0}
      {
        (\seq_item:Nn \l__braid_crossing_seq {1}
        - \seq_item:Nn \l__braid_crossing_seq {2})
        * \__braid_dim_value:n {width}
      }
      {
        (\seq_item:Nn \l__braid_crossing_seq {1}
        - \seq_item:Nn \l__braid_crossing_seq {2})
        * \__braid_dim_value:n {width}
      }
    }
  }
  {
```

```
      \fp_eval:n {
      \l__braid_height_fp + \l__braid_nudge_fp +
        \__braid_bezier_point:nnnnn
        {.5 - \__braid_value:n {gap} }
        {0}
        {\l__braid_control_fp}
        {
          \__braid_dim_value:n {height}
          - 2* \l__braid_nudge_fp
          - \l__braid_control_fp
        }
        {\__braid_dim_value:n {height} - 2* \l__braid_nudge_fp}
      }
    }

    \exp_not:N \__braid_moveto:nn
    {
      \fp_eval:n {
        (\seq_item:Nn \l__braid_crossing_seq {2} - 1)
        * \__braid_dim_value:n {width} +
        \__braid_bezier_point:nnnnn
        {.5 + \__braid_value:n {gap} }
        {0}
        {0}
        {
          (\seq_item:Nn \l__braid_crossing_seq {1}
          - \seq_item:Nn \l__braid_crossing_seq {2})
          * \__braid_dim_value:n {width}
        }
        {
          (\seq_item:Nn \l__braid_crossing_seq {1}
          - \seq_item:Nn \l__braid_crossing_seq {2})
          * \__braid_dim_value:n {width}
        }
      }
    }
    {
      \fp_eval:n {
      \l__braid_height_fp + \l__braid_nudge_fp +
        \__braid_bezier_point:nnnnn
        {.5 + \__braid_value:n {gap} }
        {0}
        {\l__braid_control_fp}
        {
          \__braid_dim_value:n {height} - 2* \l__braid_nudge_fp
          - \l__braid_control_fp
        }
        {\__braid_dim_value:n {height} - 2* \l__braid_nudge_fp}
      }
    }

    \exp_not:N \__braid_curveto:nnnnnn

    {
```

```
      \fp_eval:n {
        (.5 - \__braid_value:n {gap} ) / 3 *
        \__braid_bezier_tangent:nnnnn
        {.5 + \__braid_value:n {gap} }
        {0}
        {0}
        {
          (\seq_item:Nn \l__braid_crossing_seq {1}
          - \seq_item:Nn \l__braid_crossing_seq {2})
          * \__braid_dim_value:n {width}
        }
        {
          (\seq_item:Nn \l__braid_crossing_seq {1}
          - \seq_item:Nn \l__braid_crossing_seq {2})
          * \__braid_dim_value:n {width}
        }
      }
    }
    {
      \fp_eval:n {
        (.5 - \__braid_value:n {gap} ) / 3 *
        \__braid_bezier_tangent:nnnnn
        {.5 + \__braid_value:n {gap} }
        {0}
        {\l__braid_control_fp}
        {
          \__braid_dim_value:n {height} - 2* \l__braid_nudge_fp
          - \l__braid_control_fp
        }
        {\__braid_dim_value:n {height} - 2* \l__braid_nudge_fp}
      }
    }

    {0}
    { \fp_eval:n {
        -\l__braid_control_fp  * (.5 - \__braid_value:n {gap} )
      }
    }


    {\fp_eval:n
      {
        (\seq_item:Nn \l__braid_crossing_seq {1} - 1)
        * \__braid_dim_value:n {width}
      }
    }
    {\fp_eval:n
      {
        \l__braid_height_fp + \__braid_dim_value:n {height}
        - \l__braid_nudge_fp
      }
    }

  }
```

Now put those new strands back in the prop.

```
\prop_put:NxV \l__braid_strands_prop
{\seq_item:Nn \l__braid_crossing_seq {2}} \l__braid_tmpa_tl
\prop_put:NxV \l__braid_strands_prop
{\seq_item:Nn \l__braid_crossing_seq {1}} \l__braid_tmpb_tl
```

If the strands are more than one apart, the intermediate strands need to be broken as well.

```
\int_compare:nT
{
  \int_max:nn
  {
    \seq_item:Nn \l__braid_crossing_seq {1}
  }
  {
    \seq_item:Nn \l__braid_crossing_seq {2}
  }
  -
  \int_min:nn
  {
    \seq_item:Nn \l__braid_crossing_seq {1}
  }
  {
    \seq_item:Nn \l__braid_crossing_seq {2}
  }
  > 1
}
{
  \int_step_inline:nnnn
  {
    \int_min:nn
    {
      \seq_item:Nn \l__braid_crossing_seq {1}
    }
    {
      \seq_item:Nn \l__braid_crossing_seq {2}
    }
    + 1}
  {1}
  {
    \int_max:nn
    {
      \seq_item:Nn \l__braid_crossing_seq {1}
    }
    {
      \seq_item:Nn \l__braid_crossing_seq {2}
    }
    - 1
  }
  {

    \prop_get:NnN \l__braid_strands_prop {####1} \l__braid_tmpa_tl
    \tl_put_right:Nx \l__braid_tmpa_tl
      {
```

```
              \exp_not:N \__braid_lineto:nn
              {\fp_eval:n {(####1 - 1) * \__braid_dim_value:n {width} }}
              {\fp_eval:n
                {
                  \l__braid_height_fp + \l__braid_nudge_fp
                  + .5 * \l__braid_control_fp
                }
              }
              \exp_not:N \__braid_moveto:nn
              {\fp_eval:n {(####1 - 1) * \__braid_dim_value:n {width} }}
              {\fp_eval:n
                {
                  \l__braid_height_fp + \__braid_dim_value:n {height}
                  - \l__braid_nudge_fp - .5 * \l__braid_control_fp
                }
              }
            }

            \prop_put:NnV \l__braid_strands_prop {####1} \l__braid_tmpa_tl
          }
      }
    }
```

If we're to step the level, increase the height and add a load of coordinates.

```
    \bool_if:NT \l__braid_step_level_bool
    {
      \fp_add:Nn \l__braid_height_fp { \__braid_dim_value:n {height} }

      \int_step_inline:nnnn {1} {1} {\l__braid_strands_int}
      {
        \prop_get:NnN \l__braid_crossing_permutation_prop
        {####1} \l__braid_tmpb_tl
        \prop_get:NVN \l__braid_inverse_prop
        \l__braid_tmpb_tl \l__braid_tmpa_tl

        \__braid_coordinate:xxxx
        {-\l__braid_tmpb_tl-\int_use:N \l__braid_crossing_int}
        {-rev-\l__braid_tmpa_tl-\int_use:N \l__braid_crossing_int }
        {\fp_eval:n { (####1 - 1) * \__braid_dim_value:n {width} }}
        {\fp_to_decimal:N \l__braid_height_fp}
      }

      \int_incr:N \l__braid_crossing_int
    }
  }

  \fp_add:Nn \l__braid_height_fp
  {
    sign(\__braid_dim_value:n {height})
    * \__braid_dim_value:n {border~ height}
  }
```

Add a little bit to the end of each strand, together with some coordinates.

```
  \int_step_inline:nnnn {1} {1} {\l__braid_strands_int}
  {
```

```
    \prop_get:NxN \l__braid_strands_prop {##1} \l__braid_tmpa_tl
    \prop_get:NxN \l__braid_permutation_prop {##1} \l__braid_tmpb_tl

    \tl_put_right:Nx \l__braid_tmpa_tl {
      \exp_not:N \__braid_lineto:nn
      {\fp_eval:n { (##1 - 1) * \__braid_dim_value:n {width} }}
      {\fp_to_decimal:N \l__braid_height_fp}
      coordinate (-rev-##1-e)
      coordinate (-\l__braid_tmpb_tl-e)
      ;
    }

    \prop_put:NnV \l__braid_strands_prop {##1} \l__braid_tmpa_tl
  }
```

This is where we actually carry out the drawing commands.

```
  \int_step_inline:nnnn {1} {1} {\l__braid_strands_int}
  {
    \prop_get:NnN \l__braid_strands_prop {##1} \l__braid_tmpa_tl
    \tl_use:N \l__braid_tmpa_tl
  }
  \end{scope}
}
```

(*End definition for* `\__braid_render:`.)

`\__braid_moveto:nn` `\__braid_lineto:nn` `\__braid_curveto:nnnnnn` `\__braid_coordinate:nnnn`

These are our interfaces to the TikZ code.

```
\cs_new_nopar:Npn \__braid_moveto:nn #1#2
{
  (#1 pt, #2 pt)
}
\cs_new_nopar:Npn \__braid_lineto:nn #1#2
{
  -- (#1 pt, #2 pt)
}
\cs_new_nopar:Npn \__braid_curveto:nnnnnn #1#2#3#4#5#6
{
  .. controls +(#1 pt, #2 pt) and +(#3 pt, #4 pt)
  .. (#5 pt, #6 pt)
}
\cs_new_nopar:Npn \__braid_coordinate:nnnn #1#2#3#4
{
  \coordinate[alias=#2] (#1) at (#3 pt,#4 pt);
}
\cs_generate_variant:Nn \__braid_coordinate:nnnn {xxxx}
```

(*End definition for* `\__braid_moveto:nn` *and others.*)

`\__braid_bezier_point:nnnnn` `\__braid_bezier_tangent:nnnnn`

Used to calculate intermediate points and tangents on a bezier curve.

```
\cs_new_nopar:Npn \__braid_bezier_point:nnnnn #1#2#3#4#5
{
  \fp_eval:n
  {
    (1 - (#1)) * (1 - (#1)) * (1 - (#1)) * (#2)
    +
```

```
    3 * (1 - (#1)) * (1 - (#1)) * (#1) * (#3)
    +
    3 * (1 - (#1)) * (#1) * (#1) * (#4)
    +
    (#1) * (#1) * (#1) * (#5)
  }
}
\cs_new_nopar:Npn \__braid_bezier_tangent:nnnnn #1#2#3#4#5
{
  \fp_eval:n
  {
    3 * (1 - (#1)) * (1 - (#1)) * (#3 - (#2))
    +
    6 * (1 - (#1)) * (#1) * (#4 - (#3))
    +
    3 * (#1) * (#1) * (#5 - (#4))
  }
}
\cs_new_nopar:Npn \__braid_do_floor:nnnnn #1#2#3#4#5
{
  \pic[pic~ type=floor,
    xscale=#4,
    yscale=#5,
    at={(#2,#3)},
    braid/every~ floor/.try,
    braid/floor~#1/.try,
  ];
}
\cs_generate_variant:Nn \__braid_do_floor:nnnnn {Vxxxx}
```

(*End definition for* `\__braid_bezier_point:nnnnn` *and* `\__braid_bezier_tangent:nnnnn`.)

```
\ExplSyntaxOff
```