The **spath3** Package: Documentation

Andrew Stacey

loopspace@mathforge.org

v2.8 from 2024/05/31

Contents

1 Introduction

The `spath3` package was originally designed as a low-level package for manipulating the *soft paths* defined by PGF/TikZ. Soft paths form one stage of the stack of translations between what the author writes in the `tikzpicture` environments in their LaTeX document and what is eventually written to the output file. Most of the complicated processing has been done by the time a soft path is constructed, but it is still very definitely a TeX object and there has not, for example, been any consideration as to what the eventual output file format is (such as PDF, DVI, or SVG). So it is very amenable to being modified at this stage and this package provides a set of routines for doing so.

The original purpose was to provide a common core on which other packages would be built. Indeed, the packages `calligraphy`, `knots`, and `tilings` all use this package. However, over time I've found myself wanting to use the routines of this package at a higher level and so have designed some user-level interfaces. This document documents those.

To clarify some terminology used in this document (and more generally, this package), I regard paths as being composed of *segments* and *components*. A *segment* is a minimal drawing piece. Thus it might be a straight line or a Bézier curve. A *component* is a minimal connected section of the path. So every component starts with a move

command and continues until the next move command. For ease of implementation (and to enable a copperplate pen in the calligraphy package!), an isolated move is considered as a component.

There are no doubt bugs in this package, and useful things that I haven't implemented. If you have found one of either of these, please let me know! The best way is to open an issue at the code repository on github, at https://github.com/loopspace/spath3.

2 TikZ Keys

```
\usetikzlibrary{spath3}
```

The `spath3` TikZ library defines a set of keys that can be issued to muck about with soft paths. These are all defined in the `spath` family, so all the following keys should be prefixed by `spath/`, or the key `spath/.cd` needs to be used beforehand (but note that as yet I haven't implemented sending unknown keys back to the main `tikz` directory).

The keys try to gracefully fail if the path doesn't exist or is empty[1]. The intention is that the document should still compile with a warning in the log file (and on the console output). If this doesn't happen, please report it.

2.1 Path Names

Soft paths are stored in macros but are referred to via a name. The macro is constructed from the name, but there is scope for a prefix and suffix. By default, these are set up so to be compatible with the `intersections` library – this package and that save their paths in the same underlying macro. Some packages that are built on top of this one (such as the `knot` and `penrose` packages) install their own prefixes and suffixes to avoid clashes.

set␣prefix `set prefix={⟨prefix⟩}`
set␣suffix `set suffix={⟨suffix⟩}`
prefix   `prefix=[⟨name⟩]`
suffix   `suffix=[⟨name⟩]`
            `set name=[⟨name⟩]`

The first two keys set the prefix and suffix directly. The other keys allow for defining a label for a prefix and suffix for quick switching between different ones. The default (with key `default` or no key specified) is to match the `intersections` library. The `knots` library uses this to set a different prefix.

Using names rather than macros directly also makes it possible to link a soft path with a set of TikZ styles (this is particularly useful when splitting the path into components).

---

[1] A path that was defined inside a group will be empty if referred to outside that group rather than not existing.

`\path (0,0) -- (3,0) to[out=90, in=90] (1,0) [spath/adjust and close=current];`

There is a special path name, `current`, which refers to the current path being built. To be clear, the test for this name takes place before any prefix or suffix is applied so it depends only on what the user uses.

When this name is encountered, the current path is copied into a macro and that path is used in the command. If the command is such that it is possible that the path is modified in some fashion then after the command is finished the old current path is replaced by the new one. There are several commands that take multiple paths and `current` can be used in place of any of the paths, but usually there is only one that is modified and so that is the only one that would be used to replace the old current path.

It only makes sense to use it in a path-building context, and it refers to the path at the moment of invocation, so it doesn't do a lot in the main options on that path but it can be used in options later on in the process.

Sometimes it isn't possible to put commands partway through a path (for example, if the path command is buried in another macro) so there is a key which can be used to delay things to the end of the construction:

` end path construction={code}`

This invokes the `code` right after the path has finished being built.

2.2 Saving and Using Soft Paths

save `save=⟨name⟩`
save␣global `save global=⟨name⟩`

Saves the current path with name ⟨*name*⟩. This delays until the path is fully constructed so can be issued in the options to the main command.

Soft paths constructed this way are local to the group in which the path command is issued. The `global` version saves the path globally which is useful when the original path is inside a scope or even another tikzpicture.

If wanting to save the border of a node then because there are extra levels of grouping, the global version of this key has to be used.

clone `clone={⟨target⟩}{⟨source⟩}`
clone␣globally `clone globally={⟨target⟩}{⟨source⟩}`

Clones one soft path into another. In the second, the clone is global (the original need not be).

By using `current` as the source name, the current soft path in its current state is copied. This can be used to get round the restriction whereby `save path` waits until the path is fully constructed before being saved.

show `show=⟨name⟩`
show␣current␣path `show current path`

These keys display a path on the terminal and in the log file. They do a bit of sensible line formatting so that the paths are relatively easy to scan through.

The key `show current path` delays the logging until the path is fully constructed. To get a view of what the path is at an intermediate stage of construction, use `show=current`.

use `use=⟨name⟩`
`use={⟨name, options⟩}`

This uses a previously saved soft path at the current juncture in the path declaration. If before the path has begun, it is the initial part of the new path. If during the path construction then it is stuck in at the current place. Note that any keys that affect the soft path directly should be applied *before* this one.

The path can be modified first by using the second form – note that as far as the `use` key is concerned, the whole thing is a single argument. The options is a comma separated list and can include:

- `reverse` reverses the inserted path first.

- `weld`, `no weld` determines whether to weld the inserted path to the current path. Welding means that the `move to` at the start of the inserted path is removed. Note that this doesn't *move* the inserted path so this will usually modify the first segment of the inserted path, possibly in unexpected ways. The expected use is that when the `weld` key is used then the inserted path starts where the current path ends either because it already does or because the `move` key has also been invoked. But there's nothing *stopping* anyone using this key in other circumstances.

- `move`, `no move` determines whether the inserted path is translated so that it starts where the current path ends.

- `transform={⟨transformations⟩}` applies the specified transformations to the inserted path. See the `transform` key in Section 2.3.

- `use current transformation` applies the current transformation to the inserted path. Note that because the keys are processed and then acted upon, this key is always applied *before* the `transform` key.

- Any other key is taken as the name of the path (so it doesn't have to be specified first) with the last one winning.

One thing should be noted about transformations. By the time a soft path is built, all available transformations have been applied. This means that when re-inserting a soft path back into a high level command (such as `\draw`), existing transformations should have no effect (other than those specified via the `transform` key). Where this gets a little tricky is with things like node placement – using the `pos=D` key on a node positions that node at a particular point on the path. Exactly how the parameter is interpreted is the same as for the `spath` coordinate system described in Section 2.9. When restoring a path then the library tries to set up various internals of TikZ correctly, but there may be some things I've overlooked or not accounted for particularly with regard to existing transformations; if you spot anything working oddly then please report it to me.

These are all aliases to various versions of `use` (they originally existed as separate code before I united them all as variants of `use`). The equivalences are:

- `restore` and `insert` are aliases for `use`.

- `append` sets the `move` and `weld` keys.

- `append no move` just sets the `weld` key.

- `reverse` sets the `reverse` key.

`join with={⟨path⟩}{⟨path, options⟩}`

This joins the second path to the first, with applying the specified options. The options are the same as for the `use` key. The first path is modified, the second is not (the second path is copied before any modifications are applied).

It is almost the case that `join with={current}{...}` is synonymous with `use={...}`. The difference is that `use={...}` applies an extra check to ensure that the resulting path is not empty.

`to={⟨name⟩}`

This defines a `to` path from a soft path, so it inserts the soft path into the current path to span the gap between the start and end. The path is transformed by rotation, translation, and uniform scaling so that it exactly spans the gap required by the `to` syntax. (If the start and end point of the soft path are very close together then it won't span the gap.)

2.3 Transformation Routines

The following keys all apply some sort of transformation to the soft path. They do not render the path, but simply adjust it (but they can be used to modify the current path by invoking with the path name `current`). The global versions apply their transformation globally, otherwise it is local to the current group (or scope).

`reverse=⟨name⟩`
`reverse globally=⟨name⟩`

Reverses the soft path in place. If you want to use the original path and its reversal in the same path (for example, for constructing a region to fill) then use the `clone` key to copy it first.

`translate={⟨name⟩}{⟨x-dimen⟩}{⟨y-dimen⟩}`
`translate globally={⟨name⟩}{⟨x-dimen⟩}{⟨y-dimen⟩}`

Translates the soft path by the given dimensions.

`transform={⟨name⟩}{⟨transformations⟩}`
`transform globally={⟨name⟩}{⟨transformations⟩}`

This applies the transformation to the soft path. The transformation is processed by TikZ so should consist of TikZ-level transformations such as `shift={(2,2)}`.

span	span={⟨name⟩}{⟨start point⟩}{⟨end point⟩}
span␣global	span globally={⟨name⟩}{⟨start point⟩}{⟨end point⟩}

This transforms the named path so that it goes from the start point to the end point. As with the `to` path construction and the `splice` method, this won't work if the path ends very close to where it starts.

2.4 Intersection Routines

To use these features you need to use the `intersections` library.

split␣at␣self␣intersections	split at self intersections=⟨path⟩
split␣globally␣at␣self␣intersections	split globally at self intersections=⟨path⟩

This inserts breaks into the named soft path at the points where it intersects itself. The breaks are not gaps, to achieve that use the shortening routines after this, rather they are a change of component. Think of it as if you took the pen off the page at that point and then put it straight back down again.

split␣at␣intersections␣with	split at intersections with={⟨first⟩}{⟨second⟩}
split␣globally␣at␣intersections␣with	split globally at intersections with={⟨first⟩}{⟨second⟩}

This inserts breaks into the first path where it intersects with the second. The second path is not changed.

split␣at␣intersections	split at intersections={⟨first⟩}{⟨second⟩}
split␣globally␣at␣intersections	split globally at intersections={⟨first⟩}{⟨second⟩}

This inserts breaks into a pair of paths at their mutual intersections.

replace␣lines	replace lines=⟨path⟩
replace␣lines␣globally	replace lines globally=⟨path⟩

The PGF intersection routines have difficulties with parallel, or near parallel, lines. One way to counter this is to replace all line segments by "curves". This is done so that the parametrisation of the curve matches that of the line segment that it is replacing.

2.5 Working with Components

get␣components␣of	get components of={⟨path⟩}{⟨macro⟩}
get␣components␣of␣globally	get components of globally={⟨path⟩}{⟨macro⟩}
\getComponentOf	\getComponentOf{⟨macro⟩}{⟨number⟩}

This splits the path into a list of its components, which are stored in the macro. The macro can be used in a \foreach.

The macro consists of a comma separated list of names of the components (the actual names used are of the form `anonymous_N`). To access an individual component, use the command \getComponentOf. This can be used directly in place of a path name in any other key, such as `use`, (it is just the LaTeX3 command \clist_item:Nn).

Note that these are *copies* of the components of the original path. Changing a component doesn't update the original path.

`render␣components` render components=⟨*path*⟩

> This renders the components of a given path as separate TikZ commands, so that each can be separately styled. It applies the following styles (in this order):

1. `every spath component`

2. `spath component` ⟨*number*⟩

3. `spath component`=⟨*number*⟩

4. `every` ⟨*path*⟩ `component`

5. ⟨*path*⟩ `component` ⟨*number*⟩

6. ⟨*path*⟩ `component`=⟨*number*⟩

`insert␣gaps␣after␣components`	insert gaps after components={⟨*path*⟩}{⟨*gap*⟩}{⟨*components*⟩}
`insert␣gaps␣globally␣after␣components`	insert gaps after components={⟨*path*⟩}{⟨*gap*⟩}
`insert␣gaps␣after␣segments`	insert gaps globally after components={⟨*path*⟩}{⟨*gap*⟩}{⟨*components*⟩}
`insert␣gaps␣globally␣after␣segments`	insert gaps globally after components={⟨*path*⟩}{⟨*gap*⟩}
	insert gaps after segments={⟨*path*⟩}{⟨*gap*⟩}{⟨*segments*⟩}
	insert gaps after segments={⟨*path*⟩}{⟨*gap*⟩}
	insert gaps globally after segments={⟨*path*⟩}{⟨*gap*⟩}{⟨*segments*⟩}
	insert gaps globally after segments={⟨*path*⟩}{⟨*gap*⟩}

> This inserts a gap between components of a path by shortening the end of the specified component and start of the next one. The list of components is passed through a `\foreach` loop so that syntax like `2,4,...,16` can be used. If the list of components is not given the gaps are inserted between all components.
>
> The `segments` versions work on the segments of the path rather than its components. A *segment* is a minimal drawing piece, meaning a line or Bézier curve.

`join␣components`	join components={⟨*path*⟩}{⟨*components*⟩}
`join␣components␣globally`	join components globally={⟨*path*⟩}{⟨*components*⟩}

> This removes the `move` between each of the given components and the previous one. The list of components is processed by `\foreach`. If the component is the first one then it is joined to the last component.

join␣components␣with	join components with={⟨path⟩}{⟨splice path⟩}
join␣components␣globally␣with	join components with={⟨path⟩}{⟨splice path⟩}{⟨list⟩}
join␣components␣upright␣with	join components upright with={⟨path⟩}{⟨splice path⟩}
join␣components␣globally␣upright␣with	join components upright with={⟨path⟩}{⟨splice path⟩}{⟨list⟩}
join␣components␣with␣curve	join components with curve={⟨path⟩}
join␣components␣globally␣with␣curve	join components with curve={⟨path⟩}{⟨list⟩}
	join components globally with={⟨path⟩}{⟨splice path⟩}
	join components globally with={⟨path⟩}{⟨splice path⟩}{⟨list⟩}
	join components globally upright with={⟨path⟩}{⟨splice path⟩}
	join components globally upright with={⟨path⟩}{⟨splice path⟩}{⟨list⟩}
	join components globally with curve={⟨path⟩}
	join components globally with curve={⟨path⟩}{⟨list⟩}

This inserts the splice path in the gaps between components of path specified by a comma separated list. The splice is inserted between each specified component and the next one. If the {⟨list⟩} is not given (or is empty) then the splice is inserted between every component except that a *spot weld* is performed first to join any components where the end of one is the start of the next. The spot weld is only performed if the list of components is empty.

This does *not* close the resulting path, even if the last component is specified in the list, for that see the key close with.

Note that because there is an optional third argument to this key, the second argument must always be enclosed in braces unless it is a single token.

The upright versions do a little test to see if the gap is oriented upside-down and if so then they insert the reflection of the splice path.

The with curve versions insert a single Bézier curve into the gap. The curve is chosen to match the tangents of the segments either side of the gap and uses John Hobby's construction of a Bézier curve from its tangent directions.

spot␣weld	spot weld=⟨path⟩
spot␣weld␣globally	spot weld globally=⟨path⟩

This removes the move between any two components of the path where the end point of one component is the same as the initial point of the next (the tolerance on error here is 0.01pt).

remove␣empty␣components	remove empty components=⟨path⟩
remove␣empty␣components␣globally	remove empty components globally=⟨path⟩

This removes empty components of the path (which consist of simply a move).

remove␣components	remove components={⟨path⟩}{⟨list⟩}
remove␣components␣globally	remove components globally={⟨path⟩}{⟨list⟩}

This removes the listed components of the path. As with other list routines, the list is parsed via foreach first.

open	open={⟨path⟩}
open␣globally	open globally={⟨path⟩}

These open the path by removing all of the closing elements. If the closing elements add substantially to the path then they are replaced by a line segment.

`close`	`close={⟨path⟩}`
`close␣globally`	`close globally={⟨path⟩}`
`close␣with`	`close with={⟨path⟩}{⟨splice path⟩}`
`close␣globally␣with`	`close globally with={⟨path⟩}{⟨splice path⟩}`
`close␣with␣curve`	`close with curve={⟨path⟩}`
`close␣globally␣with␣curve`	`close globally with curve={⟨path⟩}`
`adjust␣and␣close`	`adjust and close={⟨path⟩}`
`adjust␣and␣close␣globally`	`adjust and close globally={⟨path⟩}`

These all close the last component of the given path. The first two will insert a line segment if the initial and final points of the component are not sufficiently close. The two `with` variants allow you to specify another path to insert between the end and start of the path. The two `with curve` variants insert a Bézier curve in the gap; the choice of curve depends on the tangents of the path segments being joined and uses John Hobby's Bézier curve construction (as used in the `hobby` TikZ library). The two `adjust` variants shift the end point of the path so that it is at the same place as the start of that component. If the last segment is a Bézier curve then the second control point is also shifted. (The intention with the `adjust` keys is where the start and end of the component are very close together so the adjustment creates a smaller visual effect than adding in a line – see Example 6.)

`splice`	`splice={⟨initial path⟩}{⟨splice path⟩}{⟨final path⟩}`
`splice␣global`	`splice globally={⟨initial path⟩}{⟨splice path⟩}{⟨final path⟩}`

This splices the middle path into the gap between the initial and final paths. The middle path is transformed to fit (don't try this with a path whose starting and ending points are close together) and the paths are joined so that the last component of the initial path and the first component of the splice path become a single component, and similarly at the other end.

2.6   Shortening and Splitting Paths

`shorten␣at␣end`	`shorten at start={⟨path⟩}{⟨length⟩}`
`shorten␣at␣start`	`shorten at end={⟨path⟩}{⟨length⟩}`
`shorten␣at␣both␣ends`	`shorten at both ends={⟨path⟩}{⟨length⟩}`
`shorten␣globally␣at␣end`	`shorten globally at start={⟨path⟩}{⟨length⟩}`
`shorten␣globally␣at␣start`	`shorten globally at end={⟨path⟩}{⟨length⟩}`
`shorten␣globally␣at␣both␣ends`	`shorten globally at both ends={⟨path⟩}{⟨length⟩}`

This shortens a path by the given amount from the specified end. The shortening is done so that it guarantees that it lies along the original path, but therefore the length is not completely guaranteed to be accurate. This is particularly true for Bézier paths and if there is a very short segment at the end.

It uses the derivative at the end to work out how much to shorten by. If wanting to shorten by a large amount it is better to shorten by a small amount a number of times.

split␣at	split at={⟨*path*⟩}{⟨*parameter*⟩}
split␣globally␣at	split globally at={⟨*path*⟩}{⟨*parameter*⟩}
split␣at␣into	split at into={⟨*start path*⟩}{⟨*end path*⟩}{⟨*path*⟩}{⟨*parameter*⟩}
split␣globally␣at␣into	split globally at into={⟨*start path*⟩}{⟨*end path*⟩}{⟨*path*⟩}{⟨*parameter*⟩}
split␣at␣keep␣start	split at keep start={⟨*path*⟩}{⟨*parameter*⟩}
split␣globally␣at␣keep␣start	split globally at keep start={⟨*path*⟩}{⟨*parameter*⟩}
split␣at␣keep␣end	split at keep end={⟨*path*⟩}{⟨*parameter*⟩}
split␣globally␣at␣keep␣end	split globally at keep end={⟨*path*⟩}{⟨*parameter*⟩}
split␣at␣keep␣middle	split at keep middle={⟨*path*⟩}{⟨*parameter*⟩}{⟨*parameter*⟩}
split␣globally␣at␣keep␣middle	split globally at keep middle={⟨*path*⟩}{⟨*parameter*⟩}{⟨*parameter*⟩}

These keys split a path at a point specified by a parameter. The interpretation of the parameter is described in Section 2.9.

The first two keys replace the path with the split path, so this introduces a break at the point specified by the parameter. The second two save the split path into a new path.

The ones with `keep` in the name split a path and keep the designated piece (so throw away the rest). The ones that keep the middle need two parameters to specify the break points.

arrow␣shortening	arrow shortening
	arrow shortening={⟨*true/false*⟩}

When an arrow is added to a path then the path is shortened so that the arrow tip is at the end of where the path should end up. Placing an arrow, therefore, has two effects: modifying the path and rendering the arrow. The former should happen before a soft path is manipulated, the latter afterwards. This key allows for the necessary separation to occur. Invoking it when the path is used disables the shortening at that juncture.

Note that if the path (or its last segment) is very short then when the arrow is eventually placed it may end up pointing backwards. Fixing this is on the *ToDo* list!

2.7 Exporting Paths

There are two keys to export a path.

save␣to␣aux	save to aux=⟨*path*⟩

This will save the path to the auxfile so that it is available again on the next run through.

export␣to␣svg	export to svg=⟨*path*⟩

Saves the path to the file `path.svg` as an SVG document.

2.8 Knots

knot={⟨*path*⟩}{⟨*gap*⟩}{⟨*components*⟩}
global knot={⟨*path*⟩}{⟨*gap*⟩}{⟨*components*⟩}

This style combines various of the above to make it simpler to draw knots and links. It expands to:

```
knot/.style n args={3}{
  spath/.cd,
  split at self intersections=#1,
  insert gaps after components={#1}{#2}{#3},
  maybe spot weld=#1,
  render components=#1
}
```

(The `global` version makes the path manipulating commands work globally.)

This splits a path at the points where it self-intersects and then inserts gaps between specified components. The key `maybe spot weld` does a `spot weld` depending on whether or not the key `draft mode` is set to `true` or `false`. The point here is that when designing the knot it is useful to not weld together components since that changes the component count. But once the gaps are inserted in the desired places, welding the remaining components produces a nicer diagram.

The components can be styled using the keys as described in `render components`.

2.9 Coordinates

Soft paths are not natural TikZ objects and so when replaced back into a TikZ path construction then they don't fully interact with other TikZ things, like placing nodes at points on the path (though I've done my best to make that work). To make things a little easier there is defined a coordinate system which identifies a point at a certain location along a soft path and keys which apply a transformation.

(spath cs:{⟨*name*⟩} {⟨*parameter*⟩})

The location specification is a little technical. It is specified as a number from 0 to 1, but the parameter works as follows. Let n be the number of *segments* on the path (these are the individual drawing elements that make up the path). The interval from $\frac{k-1}{n}$ to $\frac{k}{n}$ is assigned to the kth segment of the path. Then for a parameter in that interval, the location uses the natural parametrisation of that segment. For a straight line, it is simply the proportional position along but for a Bézier curve then it uses the Bézier parametrisation.

The space is vital, so if the `name` is contained in a macro then a space has to be inserted somehow. One option is to wrap the macro in braces, as seen in the examples in the next section.

`transform␣to`	`transform to={⟨path⟩}{⟨parameter⟩}`
`upright␣transform␣to`	`upright transform to={⟨path⟩}{⟨parameter⟩}`

These keys (which are in the `spath` family) set the transformation so that the origin is at the specified point of the curve (as described above) and the x–axis is tangential to the curve. The transformation is *orthogonal* in that it is achieved by a rotation and a translation.

The first key aligns the axes so that the x–axis is in the forward direction of the path as that path was constructed. The second key aligns the axes so that the y–axis points up the page. The intention with the second key is that it is similar to what happens with the `sloped` key when a node is placed on a curve.

3 Examples

1. Saving and re-using.

```
\begin{tikzpicture}
\path[spath/save=rpath] (0,0) to[out=30,in=150] (1,1);
\foreach \k in {1,...,9} {
  \fill[green] (spath cs:rpath 0.\k) circle[radius=3pt];
  \node[above left] at (spath cs:rpath 0.\k) {\(0.\k\)};
}
\fill[green] (2,2) circle[radius=3pt];
\draw[blue, spath/transform={rpath}{shift={(2,2)}},
    spath/use=rpath] node[right] {transform};
\draw[orange] (3,0) [spath/use={rpath, move}] node[right] {moving};
\draw[red] (3,-1) -- +(0,2) [spath/append=rpath] node[above]
    {append};
\draw[spath/use=rpath] node[right] {use};
\end{tikzpicture}
```
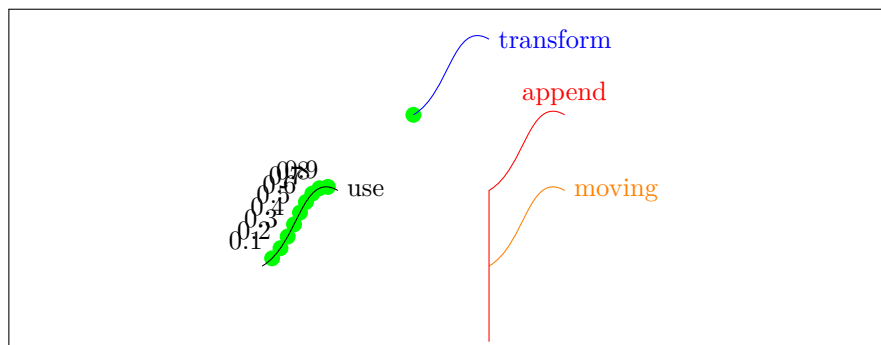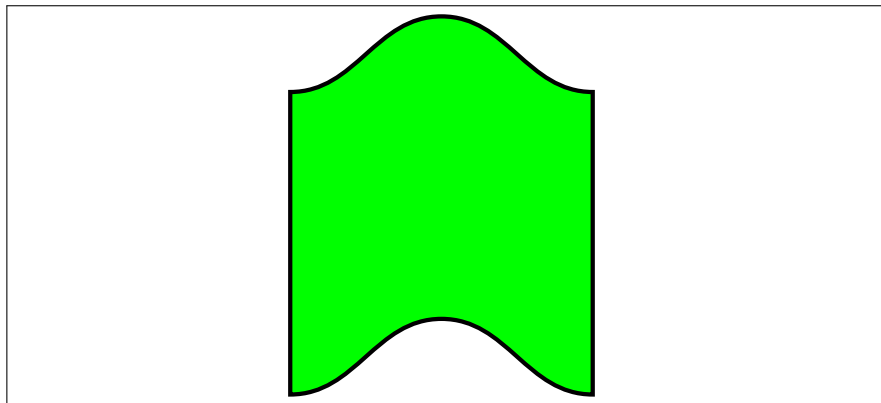

2. Reversing.

```
\begin{tikzpicture}
\path[spath/save=apath] (0,0) to[out=0,in=180] (2,1)
    to[out=0,in=180] (4,0);
\filldraw[
  green,
  draw=black,
  ultra thick,
  spath/use=apath
] -- ++(0,-4) [spath/use={apath, reverse, move, weld}] -- cycle;
\end{tikzpicture}
```
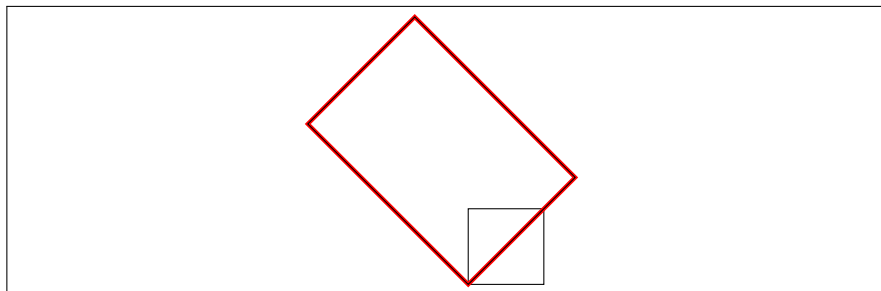

3. Transformations.

```
\begin{tikzpicture}
\draw[spath/save=tpath] (0,0) rectangle +(1,1);
\draw[rotate=45, xscale=2, yscale=3, ultra thick, red] (0,0)
    rectangle +(1,1);
\draw[
  spath/transform={tpath}{rotate=45, xscale=2, yscale=3},
  spath/use={tpath}];
\end{tikzpicture}
```
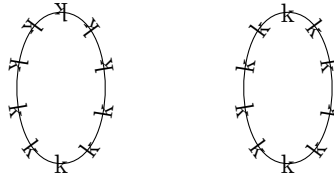
```
\begin{tikzpicture}
\draw[spath/save=oval] (0,0) to[out=0,in=0] (0,2)
    to[out=180,in=180] (0,0);
\foreach \k in {0,...,9} {
  \node[transform shape, spath/transform to={oval}{0.\k}] {k};
}
\begin{scope}[xshift = 3cm]
\draw[spath/save=soval] (0,0) to[out=0,in=0] (0,2)
    to[out=180,in=180] (0,0);
\foreach \k in {0,...,9} {
  \node[transform shape, spath/upright transform to={soval}{0.\k}]
    {k};
}
\end{scope}
\end{tikzpicture}
```
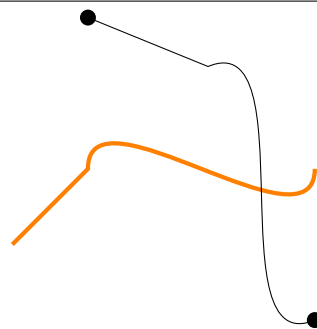


```
\begin{tikzpicture}
\path[draw=orange,ultra thick,spath/save=a] (3,2) -- ++(1,1)
    to[out=90,in=-90] ++(3,0);

\tikzset{
  spath/span={a}{(4,5)}{(7,1)}
}

\fill
(4,5) circle[radius=3pt]
(7,1) circle[radius=3pt]
;

\draw[spath/use=a];
\end{tikzpicture}
```
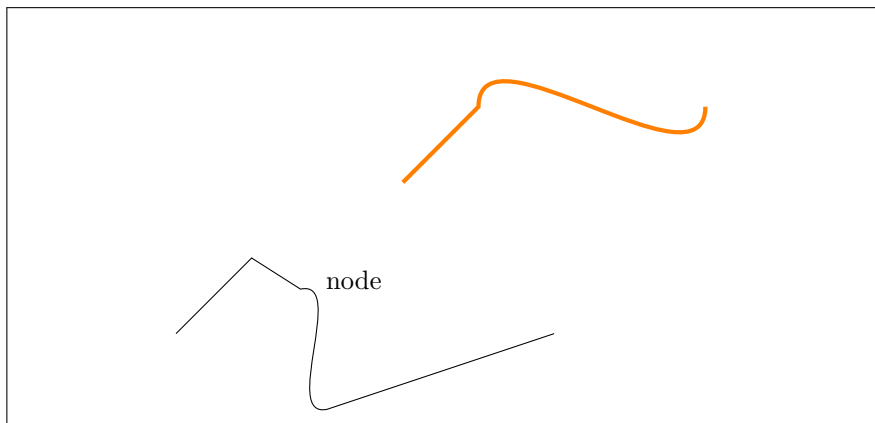
4. To paths.

```
\begin{tikzpicture}
\path[draw=orange,ultra thick,spath/save=a] (3,2) -- ++(1,1)
    to[out=90,in=-90] ++(3,0);

\draw (0,0) -- +(1,1) to[spath/to={a}] node[pos=.6,auto] {node}
    ++(2,-1) -- +(3,1);
\end{tikzpicture}
```
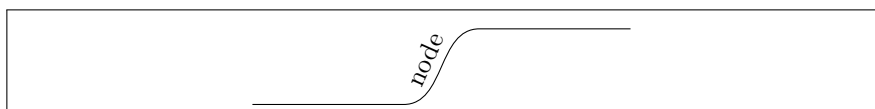

5. Node placement.

```
\begin{tikzpicture}
\path[spath/save=curve] (0,0) to[out=0,in=180] +(1,1);

\draw (0,0) -- (2,0) [spath/append=curve] node[pos=.5,auto,sloped]
    {node} -- +(2,0);

\end{tikzpicture}
```


6. Closing and adjusting paths.

```
\begin{tikzpicture}
\draw[line width=.2cm] (0,0) -- +(1,0) -- +(1,1) -- +(0,1) --
    +(.01,.01);

\draw[line width=.2cm] (2,0) -- +(1,0) -- +(1,1) -- +(0,1) --
    +(.01,.01)
-- cycle;

\draw[line width=.2cm] (4,0) -- +(1,0) -- +(1,1) -- +(0,1) --
    +(.01,.01)
[spath/close=current];

\draw[line width=.2cm] (6,0) -- +(1,0) -- +(1,1) -- +(0,1) --
    +(.01,.01)
[spath/adjust and close=current];
\end{tikzpicture}
```


7. Shortening.

```
\begin{tikzpicture}
\path[spath/save=apath] (0,0) foreach \k in {1,...,4} { -- ++(1,0)
    +(0,0)};
\draw[
  ultra thick,
  red,
  spath/.cd,
  shorten at end={apath}{7pt},
  shorten at start={apath}{9pt},
  translate={apath}{0pt}{1pt},
  use=apath,
];
\draw (0,0) circle[radius=9pt] [spath/use=apath] circle[radius=7pt];
\end{tikzpicture}
```

```
\begin{tikzpicture}
\path[spath/save=apath] (0,0) foreach \k in {1,...,4} {
    to[out=0,in=180] ++(1,0) +(0,0)};
\draw[
  ultra thick,
  red,
  spath/.cd,
  shorten at end={apath}{7pt},
  shorten at start={apath}{9pt},
  translate={apath}{0pt}{1pt},
  use=apath,
];
\draw (0,0) circle[radius=9pt] [spath/use=apath] circle[radius=7pt];
\end{tikzpicture}
```



```
\begin{tikzpicture}
\draw[spath/save=npath] (0,0) foreach \k in {1,...,4} { -- ++(1,0)
    +(0,0)};
\draw[green] (0,0) -- +(0,-3pt) foreach \k in {1,...,4} { --
    +(0,-3pt) ++(1,0)} -- +(0,-3pt);

\tikzset{
  spath/.cd,
  insert gaps after components={npath}{10pt}{1,3},
  get components of={npath}\components,
}

\tikzset{
  path 1/.style={
    red,
  },
}

\foreach[count=\k] \cpt in \components {
  \path[
    draw,
    path \k/.try,
    spath/.cd,
    translate=\cpt{0pt}{\k pt},
    use=\cpt,
  ] +(0,3pt) -- +(0,-3pt);
  \node[text=red] at (spath cs:{\cpt} .5) {\(\k\)};
}
\end{tikzpicture}
```
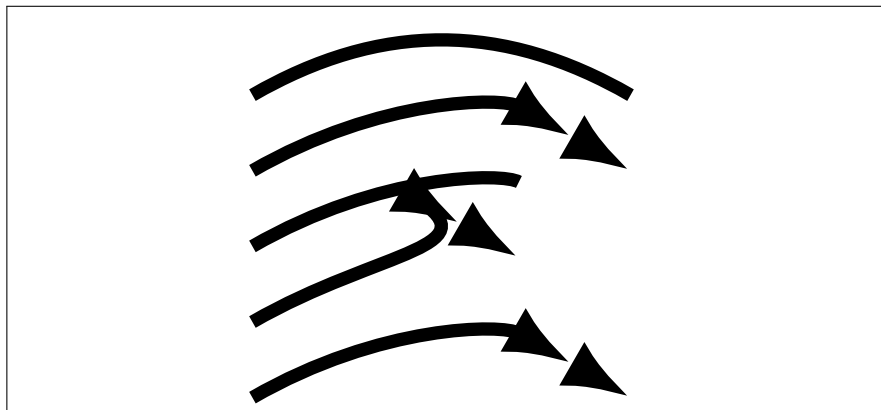

```

```
\begin{tikzpicture}[>=Latex, line width=5pt]
% Just a simple line
\draw (0,1) to[bend left] +(5,0);
% Same line but with arrows, also save the path
\draw[->.>, spath/save=arrow] (0,0) to[bend left] +(5,0);
% Let's redraw that path without the arrows - it's short! But also
    distorted
\draw[spath/use={arrow,transform={yshift=-1cm}}];
% So if we redraw it with arrows it gets doubly shortened
\draw[->.>,spath/use={arrow,transform={yshift=-2cm}}];
% If we disable the shortening, the arrows end up in the right place
\draw[->.>,spath/use={arrow,transform={yshift=-3cm}}, spath/arrow
    shortening=false];
\end{tikzpicture}
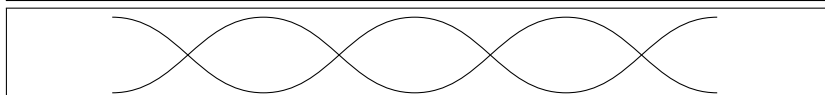```

8. Intersections.

   One of the main motivations for implementing the intersection routines was to provide a different way of drawing knots and links and similar diagrams.

   (a) Define the two paths for the braid (usually these will be defined with \path).

```
\begin{tikzpicture}[
  use Hobby shortcut,
]
\draw[spath/save global=pathA] (0,0) to[out=0,in=180] ++(2,1)
    to[out=0,in=180] ++(2,-1)  to[out=0,in=180] ++(2,1)
    to[out=0,in=180] ++(2,-1);
\draw[spath/save global=pathB] (0,1) to[out=0,in=180] ++(2,-1)
    to[out=0,in=180] ++(2,1)  to[out=0,in=180] ++(2,-1)
    to[out=0,in=180] ++(2,1);
\end{tikzpicture}
```


   (b) Split the paths at their mutual intersections and render them with a count of the components.

```
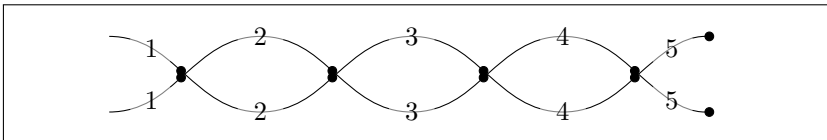\begin{tikzpicture}
\tikzset{
  spath/.cd,
  split at intersections={pathA}{pathB},
  get components of={pathA}\pathAcomponents,
  get components of={pathB}\pathBcomponents,
}

\foreach[count=\k] \cpt in \pathAcomponents {
  \draw[spath/use=\cpt,-Circle];
  \node[fill=white, fill opacity=.5, circle, text opacity=1] at
    (spath cs:{\cpt} .5) {\(\k\)};
}

\foreach[count=\k] \cpt in \pathBcomponents {
  \draw[spath/use=\cpt,-Circle];
  \node[fill=white, fill opacity=.5, circle, text opacity=1] at
    (spath cs:{\cpt} .5) {\(\k\)};
}
\end{tikzpicture}
```


(c) Now we insert gaps after certain components in each path and then render the components. To show that the gaps are genuine, we use a patterned background. Although the paths were defined globally, the splitting in the previous example was local so we need to repeat it in this one.

19

```
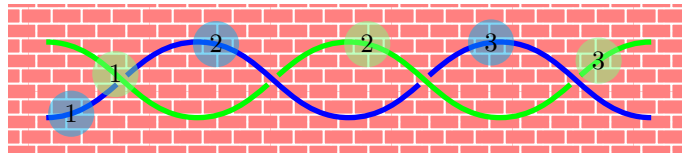\begin{tikzpicture}
\tikzset{
  spath/.cd,
  split at intersections={pathA}{pathB},
  insert gaps after components={pathA}{5pt}{1,3},
  join components={pathA}{3,5},
  get components of={pathA}\pathAcomponents,
  insert gaps after components={pathB}{5pt}{2,4},
  join components={pathB}{2,4},
  get components of={pathB}\pathBcomponents,
}

\fill[red!50!white] (-.5,-.5) rectangle (8.5,1.5);
\fill[pattern=bricks, pattern color=white] (-.5,-.5) rectangle
    (8.5,1.5);

\foreach[count=\k] \cpt in \pathAcomponents {
  \draw[blue, line width=2pt,spath/use=\cpt];
  \node[fill=cyan, fill opacity=.5, circle, text opacity=1] at
    (spath cs:{\cpt} .3) {\(\k\)};
}

\foreach[count=\k] \cpt in \pathBcomponents {
  \draw[green, line width=2pt,spath/use=\cpt];
  \node[fill=green!50, fill opacity=.5, circle, text opacity=1]
    at (spath cs:{\cpt} .3) {\(\k\)};
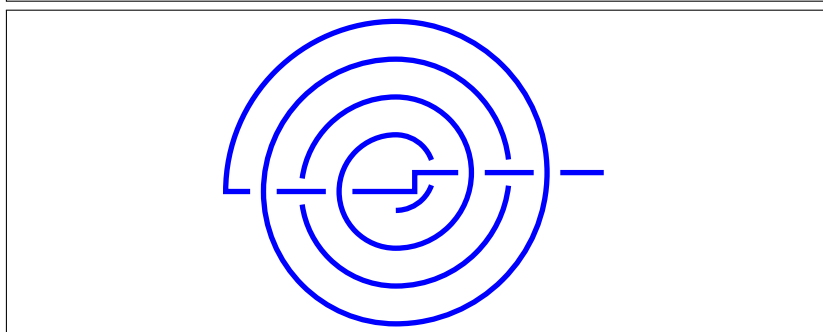}
\end{tikzpicture}
```


9. This example is notable because many of the intersection points are where segments of the path end, showing that the algorithm works well even in this circumstance.

   (a) Here's the original path.

```
\begin{tikzpicture}
\draw[spath/save global=spiral] (5,0) -- (2.5,0) -- ++(0,-.25)
    -- ++(-2.5,0)
arc[radius=2.25cm,start angle=180,end angle=90]
arc[radius=2cm,start angle=90, delta angle=-180]
arc[radius=1.75cm,start angle=-90, delta angle=-180]
arc[radius=1.5cm,start angle=90, delta angle=-180]
arc[radius=1.25cm,start angle=-90, delta angle=-180]
arc[radius=1cm,start angle=90, delta angle=-180]
arc[radius=.75cm,start angle=-90, delta angle=-180]
arc[radius=.5cm,start angle=90, delta angle=-180]
;
\end{tikzpicture}
```


(b) This renders labels on each component after splitting.

```
\begin{tikzpicture}
\tikzset{
  spath/.cd,
  split at self intersections=spiral,
  get components of={spiral}\pathcomponents,
}

\foreach[count=\k] \cpt in \pathcomponents {
  \draw[spath/use=\cpt,-|];
  \node[fill=white, fill opacity=.5, circle, text opacity=1] at
    (spath cs:{\cpt} .5) {\(\k\)};
}
\end{tikzpicture}
```

(c) Finally, we put the gaps in where we want them.

```
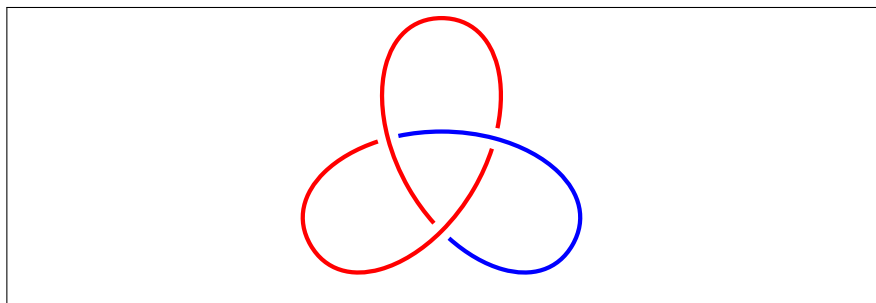\begin{tikzpicture}
\tikzset{
  spath/.cd,
  split at self intersections=spiral,
  insert gaps after components={spiral}{10pt}{1,3,5,7,10,11,14},
  spot weld=spiral,
  get components of={spiral}\pathcomponents,
}

\foreach[count=\k] \cpt in \pathcomponents {
  \draw[blue, line width=2pt,spath/use=\cpt];
}
\end{tikzpicture}
```


10. Here's a trefoil knot, demonstrating the `knot` style that simplifies creating knots.

```
\begin{tikzpicture}[
  use Hobby shortcut,
  every trefoil component/.style={ultra thick, draw, red},
  trefoil component 1/.style={blue},
]
\path[spath/save=trefoil] ([closed]90:2) foreach \k in {1,...,3} {
    .. (-30+\k*240:.5) .. (90+\k*240:2) } (90:2);
\tikzset{spath/knot={trefoil}{8pt}{1,3,5}}
\end{tikzpicture}
```


11. Here's how to mark intersections of paths with "bridges".

```
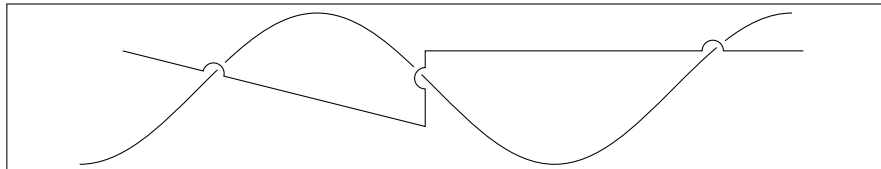\begin{tikzpicture}
\coordinate (a) at (-1,0.5);
\coordinate (b) at (8,0.5);
\coordinate (c) at (3,-0.5);
\path[spath/save=sine]
(-1.57,-1)
cos ++(1.57,1)
sin ++(1.57,1)
cos ++(1.57,-1)
sin ++(1.57,-1)
cos ++(1.57,1)
sin ++(1.57,1);
\path[spath/save=over] (a) -- (c) |- (b);

\path[spath/save=arc] (0,0) arc[radius=1cm, start angle=180, delta
    angle=-180];

\tikzset{
  spath/split at intersections with={over}{sine},
  spath/insert gaps after components={over}{8pt},
  spath/join components upright with={over}{arc},
  spath/split at intersections with={sine}{over},
  spath/insert gaps after components={sine}{4pt},
}

\draw[spath/use=sine];
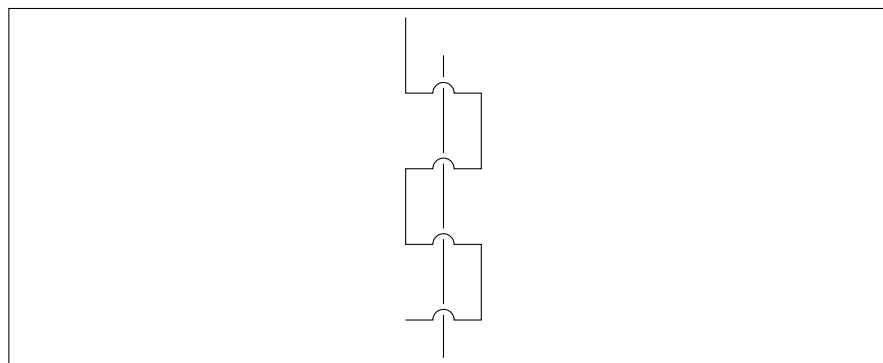\draw[spath/use=over];
\end{tikzpicture}
```


12. If there are lots of paths like the previous example, here's a convenient style to put them together.

```
\tikzset{
  bridging path/.initial=arc,
  bridging span/.initial=8pt,
  bridging gap/.initial=4pt,
  bridge/.style 2 args={
    spath/split at intersections with={#1}{#2},
    spath/insert gaps after
    components={#1}{\pgfkeysvalueof{/tikz/bridging span}},
    spath/join components upright
    with={#1}{\pgfkeysvalueof{/tikz/bridging path}},
    spath/split at intersections with={#2}{#1},
    spath/insert gaps after
    components={#2}{\pgfkeysvalueof{/tikz/bridging gap}},
  }
}

% If used in the preamble, this needs surrounding in
    \AtBeginDocument
%\AtBeginDocument{%
\tikz[overlay] { \path[spath/save global=arc] (0,0) arc[radius=1cm,
    start angle=180, delta angle=-180]; }%
%}
\begin{tikzpicture}
\path[spath/save=over] (0,0) -| ++(1,1) -| ++(-1,1) -| ++(1,1) -|
    ++(-1,1);
\path[spath/save=under] (.5,-.5) -- ++(0,4);
\tikzset{bridge={over}{under}}
\draw[spath/use=over];
\draw[spath/use=under];
\end{tikzpicture}
```


The command defining the arc does introduce a box which can introduce additional spacing. The syntax above, with the % at the end of the lines, shouldn't add space but if for some reason it does then more sophisticated solutions to this are detailed at How to save or define a TikZ path for spath3 without introducing artifacts?.