The spath3 Package: Documentation

Andrew Stacey
loopspace@mathforge.org

v2.0 from 2021/01/19

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 `penrose` 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, paths are composed of *segment* 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.

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.

2.1 Saving and Using Soft Paths

save `save=`⟨*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 soft path is actually stored in a macro constructed from the name. There are a couple of reasons for using a *name* rather than a macro directly. One is so that it is compatible with the `intersections` library – by default both this package and that save their paths in the same underlying macro. The other is to provide a way to link a soft path with a set of TikZ styles (this is particularly useful when splitting the path into components).

save␣global `save global=`⟨*name*⟩

Saves the current soft path globally.

clone `clone={`⟨*target*⟩`}{`⟨*source*⟩`}`

Clones one soft path into another.

restore `restore=`⟨*name*⟩

Restores a previously saved soft path to the current path. This happens immediately so can be issued in the options to the main command and then the path can be extended with normal drawing commands. Any keys that affect the soft path directly should be applied *before* this one.

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`), the effect of existing transformations might produce some confusing effects. When restoring a path then the library tries to set the last point correctly, but depending on how this is used it can result in transformations being applied twice.

append `append=`⟨*name*⟩

This inserts a soft path into the path at the current point, it is therefore more suited to being used part way through a path construction. In a sense, it is a little like a `pic` in that it enables the user to construct a path segment early to be reused at various places.

The path is *welded* on to the current path, meaning that it starts from the current point and there is no `move`. This is particularly useful for creating filled regions.

reverse `reverse=`⟨*name*⟩

Reverses the path in the named spath object. This doesn't do any actually drawing. The effect is local, but if you want to work with both the original path and its reversal in the same path then use the `clone` key to copy it first.

append␣reverse `append reverse=`⟨*name*⟩

Like `append` except that the inserted path is reversed first.

insert `insert=`⟨*name*⟩

Like `append` except that it doesn't move the inserted path and there is a move between the current path and the inserted path.

shorten␣at␣end
shorten␣at␣start
shorten␣at␣both␣ends `shorten at end={`⟨*name*⟩`}{`⟨*dimen*⟩`}`

Shortens the soft path by the dimension from one or both ends. This shortens the soft path along its length so that it is guaranteed that the shorter path traces a subset of the original path. For Bézier curves, however, the amount of shortening is approximate and is better for shorter distances. If wanting to shorten by a large amount it is better to shorten by a small amount a number of times.

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

Translates the soft path by the given dimensions.

transform `transform={`⟨*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)}`.

2.2 Intersection Routines

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

split␣at␣self␣intersections `split at self intersections=`⟨*path*⟩

This inserts breaks into the named soft path at the points where it intersects itself.

split␣at␣intersections `split at intersections={`⟨*first*⟩`}{`⟨*second*⟩`}`

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

2.3 Working with Components

get␣components␣of `get components of={`⟨*macro*⟩`}{`⟨*path*⟩`}`

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

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⟩}`

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`.

join␣components — `join components={⟨path⟩}{⟨components⟩}`

This removes the `move` between the given component and the previous one. If the component is the first one then it is joined to the last component.

spot␣weld — `spot weld=⟨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⟩`

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

2.4 Shortening Paths

shorten␣path␣at␣end
shorten␣path␣at␣start
shorten␣path␣at␣both␣ends — `shorten path at end={⟨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.

2.5 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.

2.6 Knots

knot `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
}
```

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`.

2.7 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. 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 `(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⟩}`

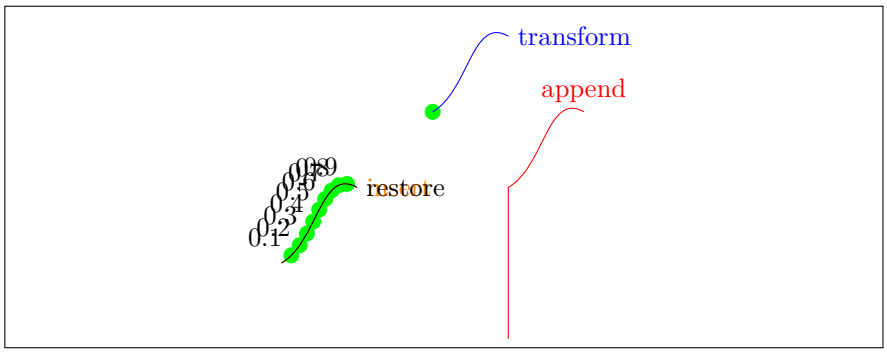
These keys (which are in the `spath` family) set the transformation so that the origin is at the specified point of the curve (as described above) and the x–axis is tangential to the curve. The transformation is *orthogonal* in that it is achieved by a rotation and a translation.

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

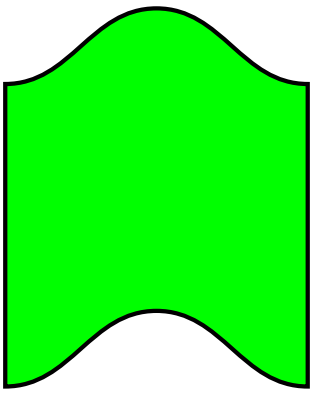
1. Saving, restoring, inserting, and appending.

```
\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/restore=rpath] node[right] {transform};
\draw[orange] (3,0) [spath/insert=rpath] node[right] {insert};
\draw[red] (3,-1) -- +(0,2) [spath/append=rpath] node[above]
    {append};
\draw[spath/restore=rpath] node[right] {restore};
\end{tikzpicture}
```



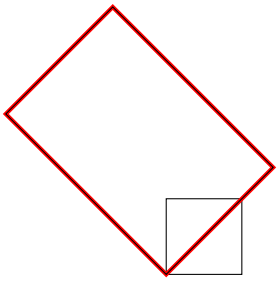
2. Reversing.

```
\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/restore=apath
] -- ++(0,-4) [spath/append reverse=apath] -- cycle;
\end{tikzpicture}
```

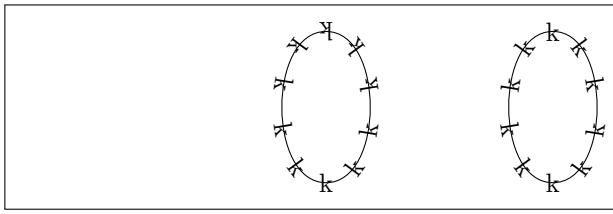


3. Transformations.

```
\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/restore={tpath}];
\end{tikzpicture}
```

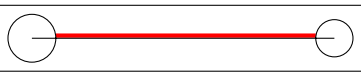


```
\begin{tikzpicture}
\draw[spath/save=oval] (0,0) to[out=0,in=0] (0,2)
    to[out=180,in=180] (0,0);
\foreach \k in {0,...,9} {
  \node[transform shape, spath/transform to={oval}{0.\k}] {k};
}
\begin{scope}[xshift = 3cm]
\draw[spath/save=soval] (0,0) to[out=0,in=0] (0,2)
    to[out=180,in=180] (0,0);
\foreach \k in {0,...,9} {
  \node[transform shape, spath/upright transform to={soval}{0.\k}]
    {k};
}
\end{scope}
\end{tikzpicture}
```



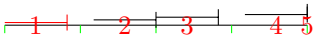
4. Shortening.

```
\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},
  restore=apath,
];
\draw (0,0) circle[radius=9pt] [spath/insert=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},
  restore=apath,
];
\draw (0,0) circle[radius=9pt] [spath/insert=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},
    restore=\cpt,
  ] +(0,3pt) -- +(0,-3pt);
  \node[text=red] at (spath cs:{\cpt} .5) {\(\k\)};
}
\end{tikzpicture}
```

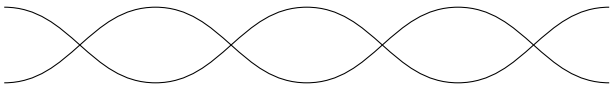


5. Intersections.

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

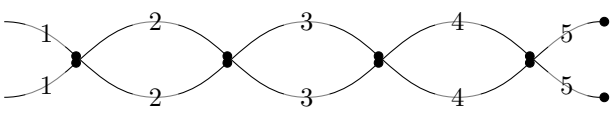
   (a) Define the two paths for the braid (usually these will be defined with `\path`).

```
\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.

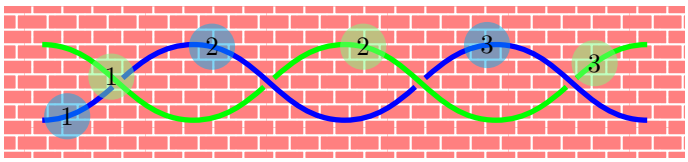
```
\begin{tikzpicture}
\tikzset{
  spath/.cd,
  split at intersections={pathA}{pathB},
  get components of={pathA}\pathAcomponents,
  get components of={pathB}\pathBcomponents,
}

\foreach[count=\k] \cpt in \pathAcomponents {
  \draw[spath/restore=\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/restore=\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.

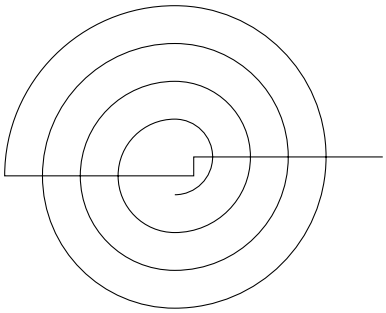
```
\begin{tikzpicture}
\tikzset{
  spath/.cd,
  split at intersections={pathA}{pathB},
  insert gaps after components={pathA}{5pt}{1,3},
  join components={pathA}{3,5},
  get components of={pathA}\pathAcomponents,
  insert gaps after components={pathB}{5pt}{2,4},
  join components={pathB}{2,4},
  get 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/restore=\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/restore=\cpt];
  \node[fill=green!50, fill opacity=.5, circle, text opacity=1]
    at (spath cs:{\cpt} .3) {\(\k\)};
}
\end{tikzpicture}
```



6. 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.

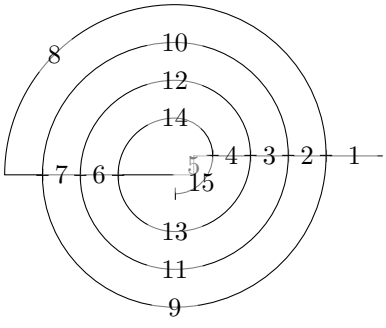
   (a) Here's the original path.

```
\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}
```



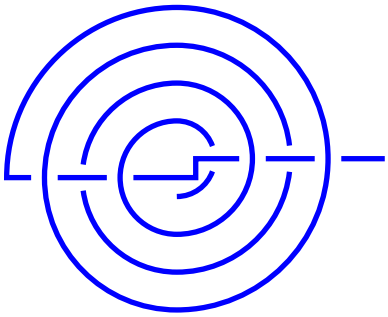
(b) This renders labels on each component after splitting.

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

\foreach[count=\k] \cpt in \pathcomponents {
  \draw[spath/restore=\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/restore=\cpt];
}
\end{tikzpicture}
```



7. 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}
```