The tikz-nfold package

Jonathan Schulz

March 2023

Abstract

This package provides an alternative to TikZ' `/tikz/double` option, avoiding some shortcomings of the original approach. It also provides an option to draw triple, quadruple, and n-fold paths.

Compatibility

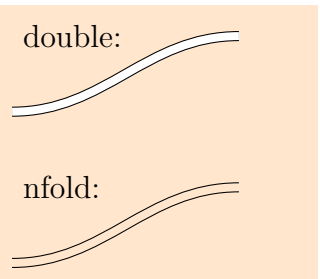
This package has been tested with `pdflatex`, `lualatex` and `xelatex`. Support for plain TEX could, in principle, be implemented in the future as well.

1 Quick start

Add

```
\usetikzlibrary{nfold}
```

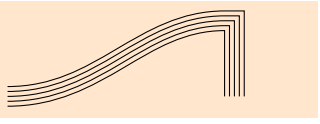
to your preamble. Now you can add the style `/tikz/nfold` to any path that uses `/tikz/double`:




```
\begin{tikzpicture}
  \draw[double distance=3pt]
    (0,2) to[out=0, in=180] (3,3);
  \draw[double distance=3pt, nfold]
    (0,0) to[out=0, in=180] (3,1);
  \node[right] at (0,3) {double:};
  \node[right] at (0,1) {nfold:};
\end{tikzpicture}
```

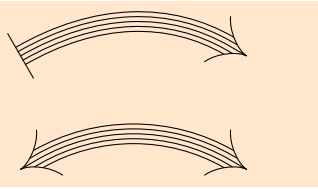
While it appears that adding `nfold` does not do much here, it avoids some rendering issues of `/tikz/double`, hence I recommend using it in most cases (see Section 2.1 for details).

Specify a number for n-fold lines:



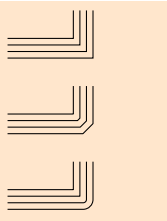
```
\begin{tikzpicture}
  \draw[double distance=7pt, nfold=5]
    (0,0) to[out=0, in=180] (3,1) -- (3,0);
\end{tikzpicture}
```

The arrow tips `Implies` and `Bar` are supported (the latter can also be aliased by `|`):



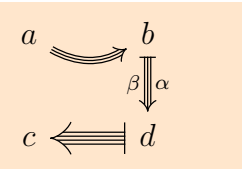
```
\begin{tikzpicture}
  \draw[double distance=7pt, nfold=5, arrows=Bar-Implies]
    (0,1.5) to[bend left] (3,1.5);
  \draw[double distance=7pt, nfold=5, arrows=Implies-Implies]
    (0,0) to[bend left] (3,0);
\end{tikzpicture}
```

Different line joins are supported:



```
\begin{tikzpicture}[line join=bevel]
  \draw[line join=miter, double distance=7pt, nfold=4]
    (0,2) -- (1, 2) -- (1,2.5);
  \draw[double distance=7pt, nfold=4]
    (0,1) -- (1, 1) -- (1,1.5);
  \draw[line join=round,  double distance=7pt, nfold=4]
    (0,0) -- (1, 0) -- (1,.5);
\end{tikzpicture}
```

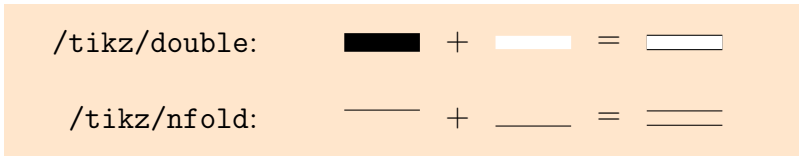
There is also support for `tikz-cd`:




```
\begin{tikzcd}
  a \ar[r, Rightarrow, bend right, nfold=3] &
  b \ar[d, Mapsto, nfold=3, "\alpha", "\beta"'] \\
  c \ar[r, Mapsfrom, double distance=4pt, nfold=4] &
  d
\end{tikzcd}
```

2 Comparison to `/tikz/double`

This package does *not* aim to supersede `/tikz/double`, as both the original and the `nfold` approach have their own strengths and weaknesses. The main difference is that `/tikz/double` achieves its goal by drawing the original path twice, once very thick with the foreground colour and then slightly less thick with the background colour. By contrast, `nfold` offsets the path:



(1)

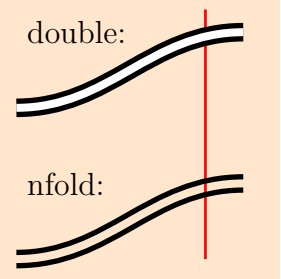
2.1 Issues with `/tikz/double`

While the approach of `/tikz/double` is very robust and efficient, it does have a few pitfalls:

- Different types of visual glitches can occur in PDF renderers:
  - One common issue is that the white foreground piece completely covers the black background piece at certain zoom levels, leading to the top or bottom part of the doubled path missing (depending on your PDF viewer and zoom level, this issue might be visible in Eq. (1)).
  - Another common glitch is the appearance of a thin horizontal line at the start and end of the doubled path (visible in most examples of curved paths if your

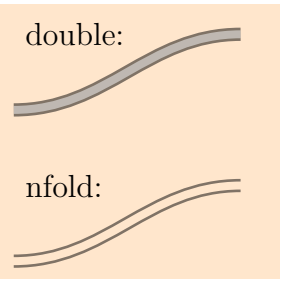
viewer has this problem). The reason is that the larger black path in the background is not perfectly covered by the smaller white foreground piece, most likely due to rounding errors.

- The approach assumes that the background has a uniform colour, and it is the user's responsibility to correctly set the background colour:




```
\begin{tikzpicture}
  \draw[red, line width=1pt] (2.5, 3.3) -- (2.5, 0);
  \draw[line width=2pt, double distance=3pt]
    (0,2) to[out=0, in=180] (3,3);
  \node[right] at (0,3) {double:};
  \draw[line width=2pt, double distance=3pt, nfold]
    (0,0) to[out=0, in=180] (3,1);
  \node[right] at (0,1) {nfold:};
\end{tikzpicture}
```

- Transparency does not work correctly:



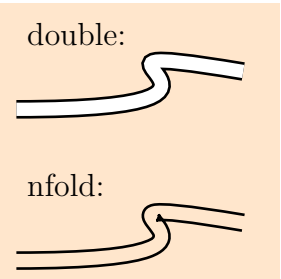

```
\begin{tikzpicture}[line width=1pt]
  \draw[double distance=3pt, opacity=.5]
    (0,2) to[out=0, in=180] (3,3);
  \draw[double distance=3pt, opacity=.5, nfold]
    (0,0) to[out=0, in=180] (3,1);
  \node[right] at (0,3) {double:};
  \node[right] at (0,1) {nfold:};
\end{tikzpicture}
```

- Triple and n-fold paths are not supported (although this could be implemented in principle).

2.2 Issues with `/tikz/nfold`

This package is far from perfect, and even if it were, there would still be some cases where the approach of `/tikz/double` is better suited. Here are some shortcomings of `nfold`:

- `nfold` struggles with high curvatures and wide paths: Let $\kappa(t)$ be the curvature of the path in a given point, and let `double distance` $= \alpha$. If $\kappa(t) > \frac{2}{\alpha}$ (i.e. the radius of the osculating circle is smaller than half the width of the path) for some $0 \le t \le 1$, the output of `nfold` will not be correct:




```
\begin{tikzpicture}
  \draw[double distance=5pt, line width=1pt]
    (0,2) .. controls (4,2) and (0,3) .. (3,2.5);
  \node[right] at (0,3) {double:};
  \draw[double distance=5pt, line width=1pt, nfold]
    (0,0) .. controls (4,0) and (0,1) .. (3,.5);
  \node[right] at (0,1) {nfold:};
\end{tikzpicture}
```

  Some, but not all of these cases raise warnings (this feature is on the wish list).

- Some rare cases of curves are not offset correctly. The reasons for that are discussed below in Appendix A.4. Usually, slightly changing the control points or values of the curve will fix the problem.

- Closing paths (i.e. using `-- cycle`) is not yet supported (this feature is on the wish list).
- `nfold` is significantly slower than `/tikz/double`. Part of the reason is that the construction is far more complex, other reasons can be fixed in principle. Specifically, the use of the `decorations` library is rather inefficient for this purpose.
- While I did my best trying to break `nfold` with as many of TikZ' options as possible, there are most definitely some problematic options I have not tested. If you find any bugs, please report them here.

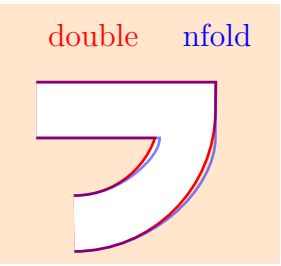
3 Known issues

3.1 Fixable / wish list

- Closing paths (e.g. `-- cycle`) is not yet properly supported.
- The arrow detection is still quite rough and likely has some bugs. One known issue is that you cannot provide parameters to the arrow tips (e.g. `Implies[red]`).
- Similarly, redefining arrows does not have the desired effect (usually, it has no effect at all).
- It would not be very hard to check for too much curvature in the offsetting algorithm and throw warnings in these cases.
- Migrating away from the `decorations` library and integrating this algorithm more tightly with the rendering pipeline would be possible. I expect that doing so will significantly improve the performance and also fix most arrow-related issues. This idea is on the back burner for now, as it likely requires changes to the TikZ rendering pipeline. If the TikZ team shows interest in integrating this library, I will reconsider this.
- Discontinuous paths with an arrow tip at the start are rendered differently in `/tikz/double` and `nfold`. As I do not really see a use case for such paths (and a rendering pipeline integration would likely fix it anyway), this issue is not a priority for now.

3.2 Impossible or very hard to fix

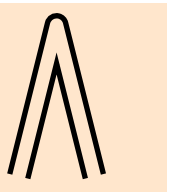
- Correctly rendering paths with too much curvature is borderline impossible with this approach. This is one of the cases where using `/tikz/double` is the only viable option.
- Curves of `nfold` slightly deviate from the curves of `/tikz/double` near joins with a non-zero angle:




```
\begin{tikzpicture}[line width=1pt]
  \draw[red, double distance=20pt]
    (0,2) -- (2,2) to [out=-90, in=0] (.5,.5);
  \draw[blue, double distance=20pt, opacity=.5, nfold]
    (0,2) -- (2,2) to [out=-90, in=0] (.5,.5);
  \node[right, red] at (0,3) {double};
  \node[left, blue] at (3, 3) {nfold};
\end{tikzpicture}
```

This cannot be fixed without extensive use of the `intersections` library, hurting the performance, and the result might still not look great for orders ≥ 3.

- changing joins in `\pgfsys@beginscope` without an accompanying TEX group may cause inconsistent behaviour in the joins. For example,



```
\makeatletter
\begin{tikzpicture}[line join=miter, line width=2pt]
  \pgfsys@beginscope
    \pgfsetroundjoin
  \pgfsys@endscope
  \draw[double distance=5pt, nfold] (0,0) -- (.5,2) -- (1,0);
\end{tikzpicture}
\makeatother
```

has `round` joins on the large path, but `miter` joins on the constituent paths. This problem does not occur with `\pgfscope`.

4 The basic layer pgf commands

This package also provides some basic layer commands for offsetting curves and straight lines. Use

`\usepgflibrary{bezieroffset}`

to only import the base layer library. The following commands are provided:

- `\pgfoffsetcurve`: This macro draws the parallel of a Bézier curve. It takes five parameters, the first four being the four control points of the Bézier curve (e.g. in the form of `\pgfpoint{}{}`), the fifth parameter is the distance by which the curve should be offset. A negative value offsets the curve in the opposite direction. This macro begins with a `\pgfpointmoveto` to the offset first control point.
- `\pgfoffsetcurvenomove`: The only difference to the previous macro is that this version does not move to the offset first control point. This is useful if one wants to offset an uninterrupted path consisting of several curves. The output will only be correct if the previous path segment ends on the offset first control point.
- `\pgfoffsetline`: This macro offsets a straight line. It takes two points and the distance as parameters.
- `\pgfoffsetlinenomove`: This macro is analogous to `\pgfoffsetcurvenomove`.

A The offsetting algorithm

This algorithm is based on an algorithm by Pomax. See A Primer on Bézier curves, the source code can be found here.

A.1 Simple and fully simple Bézier curves

Throughout this section the term "Bézier curve" refers to a cubic Bézier curve, which is defined by four points (A_1, A_2, A_3, A_4).

As explained in the aforementioned source, in almost all cases the parallel of a Bézier curve is not exactly a Bézier curve itself. To approximate the parallel using Bézier curves, we therefore first divide the given curve into "simple" segments which can be offset with reasonable accuracy. The following defines a simple segment:

Definition A.1. A Bézier curve (A_1, A_2, A_3, A_4) is *simple* if

1. the points A_2 and A_3 lie on the same side of the line $\overline{A_1A_4}$,
2. the absolute angle between the tangents in A_1 and A_4 is at most $\pi/3$ (i.e. the cosine is no smaller than 0.5), and
3. the distances fulfil $\overline{A_1A_2} + \overline{A_3A_4} \leq \overline{A_1A_4}$.[1]

Definition A.2. A Bézier curve (A_1, A_2, A_3, A_4) is *fully simple*[2] if all of its segments are simple in the sense of Definition A.1.

In order to offset an arbitrary Bézier curve we split it into fully simple segments.

A.2 Subdivision

It is well known that at every point $0 < t < 1$, a Bézier curve $A = (A_1, A_2, A_3, A_4)$ can be subdivided into two Bézier curves B and C using de Casteljau's algorithm (which naturally fulfil $A_1 = B_1$ and $A_4 = C_4$). A more or less heuristic fact is that B and C are "more likely" to be simple than A (if you can prove any of the statements here, please contact me). Hence, if one wants to offset a non-simple curve A, one could try to subdivide A until all of its segments are simple, then offset each segment.

[1] The reference only uses the first two conditions.

[2] This terminology is not used in the source.

A.3 Pomax' approach

The original approach by Pomax consists of two passes. The first pass subdivides A on all extrema in x or y. In a second pass, each segment $A^{(i)}$ is made simple in steps of $t \mapsto t + 0.01$, roughly using the following pseudocode:

```
t_1 = t_2 = 0.0
while t_2 < 1.0:
  S = segment(A from t_1 to t_2+0.01)
  if not isSimple(S):
    segments += [S]
    t_1 = t_2
  t_2 += 0.01
```

Essentially, this verifies with great certainty that the segment is fully simple in the sense of

The main reason this approach is not used in this library is performance, as the library is slow enough already. Other minor reasons include that the original approach is not invariant under reversals or rotations: Reversing and/or rotating a curve yields a different subdivision and hence potentially a slightly different-looking curve.

A.4 The approach used here

In this library, we instead take a recursive approach:

```
def split(A, level):
  if isSimple(A):
    segments += [A]
  else:
    if level < 0:
      Display a warning
      segments += [A]
    else:
      first, second = split(A, t=0.5)
      split(first, level-1)
      split(second, level-1)
```

The default maximum depth is 5, so the curve is split into at most $2^5 = 32$ segments. This has the downside that some simple but not fully simple curves may remain undetected and be offset slightly incorrectly. If you encounter examples of such curves with bad outputs or if you have any ideas for additional constraints to add to Definition A.1 that can be checked with reasonable computational effort, please be in touch.

A.5 Offsetting simple Bézier curves

Disregarding edge cases (which will be discussed later), offsetting the curve works as follows:

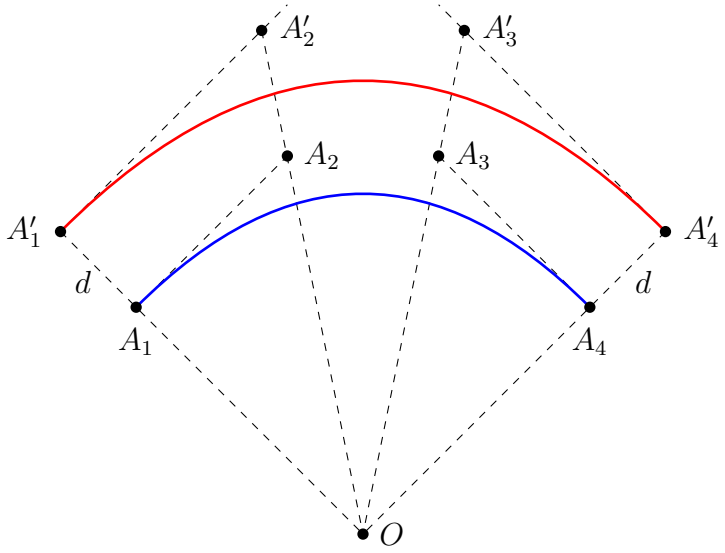
1. Construct lines orthogonal to the tangent in A_1 and A_4 and find their intersection. This point is called the *origin* of the curve.

2. The new control points A_1' and A_4' are given by A_1 and A_4 offset orthogonally to the tangent.

3. Construct a ray from A_1' parallel to the tangent in A_1, and construct another ray from the origin through A_2. Now A_2' is given by the intersection of those rays.

4. A_3' can be constructed similarly.

The construction is shown in the following picture:

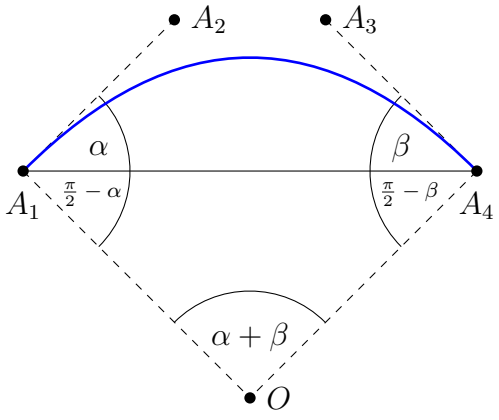


A.6 Removing singularities

Clearly, as the angle between $\overrightarrow{A_1A_2}$ and $\overrightarrow{A_3A_4}$ decreases, the origin approaches infinity. Computing the control points (A_i') by computing the coordinates of O is therefore not numerically stable for almost straight curves. However, we can get around this problem using elementary geometry. Constructing A_1' and A_4' is independent of the location of the origin, so the only difficult part is computing the distances $\overline{A_1'A_2'}$ and $\overline{A_3'A_4'}$. We find

$$\frac{\overline{A_1'A_2'}}{\overline{A_1A_2}} = \frac{\overline{OA_1'}}{\overline{OA_1}} = 1 + \frac{d}{\overline{OA_1}} \implies \overline{A_1'A_2'} = \overline{A_1A_2}\left(1 + d \cdot \frac{1}{\overline{OA_1}}\right) \tag{2}$$

which is regular as O approaches infinity. Now we to determine the inverse of $\overline{OA_1}$, which can be computed using the law of sines:



$$\frac{\sin(\frac{\pi}{2}-\alpha)}{\overline{OA_4}} = \frac{\sin(\alpha+\beta)}{\overline{A_1A_4}} = \frac{\sin(\frac{\pi}{2}-\beta)}{\overline{OA_1}} \implies \frac{1}{\overline{OA_1}} = \frac{1}{\overline{A_1A_4}} \cdot \frac{\sin(\alpha+\beta)}{\cos(\beta)} \,. \tag{3}$$

Note that $-\frac{\pi}{2} < \beta < \frac{\pi}{2}$ (and hence $\cos(\beta) > 0$) is guaranteed if the curve is simple — in fact, simplicity guarantees $\alpha \cdot \beta > 0$ and $|\alpha| + |\beta| \leq \frac{\pi}{3}$. Using the sine addition theorem we can further rewrite the fraction to

$$\frac{\sin(\alpha+\beta)}{\cos(\beta)} = \sin(\alpha) + \cos(\alpha)\frac{\sin(\beta)}{\cos(\beta)} \,, \tag{4}$$

and all of these terms can be computed directly from dot and cross products of vectors between the original control points. To summarise, let $\vec{v}_{ij} := \overrightarrow{A_iA_j}$, and let $\vec{v}_0$, $\vec{v}_1$ be the normalised tangents at $t = 0$ and $t = 1$, respectively (see Appendix A.8 how they are computed). Then

$$\overline{A_1'A_2'} = \overline{A_1A_2} + \frac{d}{\overline{A_1A_4}^2} \cdot \left(\vec{v}_{12} \times \vec{v}_{14} - \vec{v}_{12} \cdot \vec{v}_{14} \frac{\vec{v}_{14} \times \vec{t}_1}{\vec{v}_{14} \cdot \vec{t}_1} \right) \,. \tag{5}$$

Let furthermore $\vec{u}_{ij} := \vec{v}_{ij}/|\vec{v}_{ij}|$ be the normalised vectors. Then we find

$$\begin{aligned} \overline{A_1'A_2'} &= \overline{A_1A_2} + \frac{d}{\overline{A_1A_4}} \left(\vec{v}_{12} \times \vec{u}_{14} - \vec{v}_{12} \cdot \vec{u}_{14} \frac{\vec{u}_{14} \times \vec{t}_1}{\vec{u}_{14} \cdot \vec{t}_1} \right) \,, \\ \overline{A_4'A_3'} &= \overline{A_4A_3} + \frac{d}{\overline{A_1A_4}} \left(\vec{v}_{43} \times \vec{u}_{14} - \vec{v}_{43} \cdot \vec{u}_{14} \frac{\vec{u}_{14} \times \vec{t}_0}{\vec{u}_{14} \cdot \vec{t}_0} \right) \,. \end{aligned} \tag{6}$$

A.7 Edge cases

A.7.1 Overlaps: $A_i = A_{i+1}$

If there is one overlap $A_1 = A_2$, $A_2 = A_3$ or $A_3 = A_4$, the cubic Bézier curve reduces to a quadratic one. For two overlaps, we get a linear Bézier curve (i.e. a straight line), and for three overlaps we get a point. The main problem to watch out for is that the tangents $\vec{t}_0$ and $\vec{t}_1$ need to be computed differently:

- If $A_1 \neq A_2$, we find $\vec{t}_0 = \vec{u}_{12}$.
- If $A_1 = A_2 \neq A_3$, we find $\vec{t}_0 = \vec{u}_{13}$.[3]
- If $A_1 = A_2 = A_3 \neq A_4$, we find $\vec{t}_0 = \vec{u}_{14}$.
- If $A_1 = A_2 = A_3 = A_4$, the curve is just a point and the tangent is not defined. The implementation defaults to $\vec{t}_0 = (1, 0)$.

The analogous statement hold for $\vec{t}_1$. In practice we test for approximate, not exact equality.

A.7.2 Overlaps $A_1 = A_4$

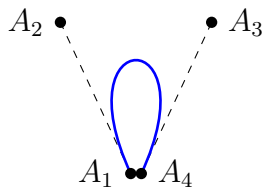
Equation (6) has one remaining singularity, namely for $A_1 \approx A_4$. This singularity is fundamental and not an artefact: As A_1 approaches A_4 while $\overline{A_1A_2}$ and $\overline{A_3A_4}$ stay constant,

[3] This implicitly assumes a regular reparametrisation of the curve (e.g. a parametrisation over arc length); the usual parametrisation has a gradient of zero at $t = 0$.

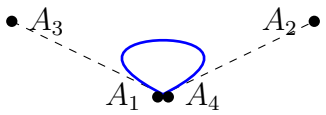
O also approaches A_4, hence the angle between $\overrightarrow{A_1A_2}$ and $\overrightarrow{OA_2}$ approaches zero, sending the intersection point A_2' to infinity.

A closer inspection of this special case reveals a number of sub-cases:

- Fully degenerate curves with $A_1 \approx A_2 \approx A_3 \approx A_4$. Offsetting these curves in a stable manner is impossible, as the gradient and hence the direction in which to offset is numerically unstable. Rendering such curves will hardly have any output at all anyway. See below in Appendix A.8 how this case is handled.
- Non-trivial loops with $\overline{A_1A_2} \gg \overline{A_1A_4}$ and/or $\overline{A_3A_4} \gg \overline{A_1A_4}$, $|\alpha| + |\beta| \geq \frac{\pi}{3}$. Such curves violate the second and third condition of Definition A.1, for example:



- Curves with $\overline{A_1A_2} \gg \overline{A_1A_4}$ and/or $\overline{A_3A_4} \gg \overline{A_1A_4}$, $|\alpha|+|\beta| < \frac{\pi}{3}$: Such curves violate the third condition of Definition A.1, for example:



A.8 Stabilising the offsetting algorithm for non-simple curves

As seen in Appendix A.4, there are cases where the offsetting algorithm will be called on non-simple curves. In such cases it is essential that the code does not crash (e.g. from division by zero), and that the output produced is at least somewhat sensible. The following measures are taken:

- If $\overline{A_1A_4}$ is very close to zero,[4] we set $\overline{A_1'A_2'} = \overline{A_1A_2}$ and $\overline{A_3'A_4'} = \overline{A_3A_4}$, forming a rather rough approximation of the parallel curve. This causes a warning to be logged.
- For simple curves we find $\vec{u}_{14} \cdot \vec{u}_{34} \geq 0.5$ in the denominator. Therefore, for non-simple curves, the denominator is clamped to $[0.5, 1.0]$, preventing division by zero.

[4] Clamping the fraction $d/\overline{A_1A_4}$ to some maximum value did not work well, as it had a tendency of producing false positives.