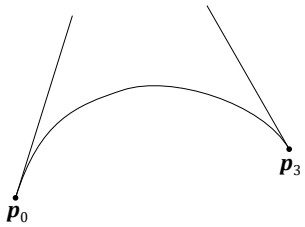
A simplifying plugin

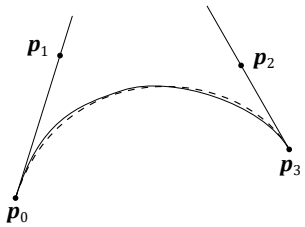
The problem: How to approximate a curve with a Bézier curve

Originally I was not going to make any path-modifying plugins. What I had in mind was to find a better algorithm to approximate a general curve as a composite Bézier curve. I already have an algorithm of my own, and I have based several plugins on it: Parametric curves, Path transformations, Path warping, and possibly others. And I am not at all satisfied with that rickety approximation algorithm. Too complicated, too slow, too unreliable, and so on. So, when I had some extra time in my hands I set out to find better.

The basic problem is as follows. Assume we have a regular curve, for example a parametric curve, given as a mathematical expression $\boldsymbol{r} = \boldsymbol{f}(t)$, $a \leq t \leq b$. Denote $\boldsymbol{p}_0 = \boldsymbol{f}(a)$, $\boldsymbol{p}_3 = \boldsymbol{f}(b)$. Assume that we know the tangents at $\boldsymbol{p}_0$ and $\boldsymbol{p}_3$.



We want to draw a Bézier arc approximating the curve. And we want it to have the same end points and the same tangents there. We need four control points $\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3$, of which we already know $\boldsymbol{p}_0$ and $\boldsymbol{p}_3$, and the tangent condition implies that $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ must be on the tangents.

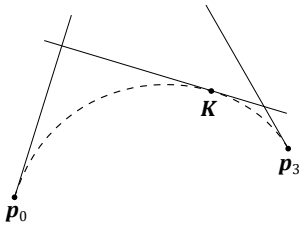


The dashed line is one such Bézier arc. So, we have two unknowns (the handle lengths), hence we need two equations. If we have such equations then in principle we should be able to solve the handle lengths.

All this is very basic. The interesting question is, from where do we get the two equations? Which conditions should we use? I have tried some. What they are and what my current rickety algorithm is I rather leave unexplained. Let us forget this and go to a new idea.

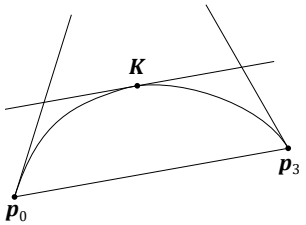
New idea for the problem of approximating a curve

Pondering different possibilities I realized that if we require that the Bézier arc runs through a pre-defined point K and has there a pre-defined tangent, then this leads to a group of three equations with three unknowns which can be solved. Deriving the formulas (not shown here) was the main mathematical hurdle (not hard).



Two of the unknowns are the handle lengths, and the third is the parameter value t at which K is reached. Applying the formulas means solving one polynomial equation of degree three, otherwise it is easy.

This gave an idea of a simple algorithm to solve the problem: First find on the original curve the point K farthest from the chord. Then the curve has at K normally a tangent parallel to the chord.



Then, using the derived formulas, find Bézier arcs which have these particular three tangents at the three points $\boldsymbol{p}_0$, $\boldsymbol{p}_3$, K. There may be three solutions. If all is well, one of them is a good approximation of the original curve. If not, then use K to split the curve in two: take arcs $\boldsymbol{p}_0K$ and $K\boldsymbol{p}_3$. Then try the algorithm for each half. And so on, recursively.

This is a neat idea, at least when compared with some I have tried previously. But when one starts implementing, problems arise, such as:

- Some special cases need special treatment: the case when the tangents at the end points are parallel; the case $\boldsymbol{p}_0 = \boldsymbol{p}_3$; and some others.
- It often appears that none of the 1–3 Bézier arcs is acceptable but if one allows some variation in the point K or in the direction of the tangent at K then an acceptable arc is found. Therefore, the code had better contain

loops running through some such variations prior to resorting to splitting the arc.

- One needs some routine to measure how much a candidate Bézier arc differs form the original curve. This routine appeared to be the biggest culprit to grow the running time since it is called very often. So the routine must be simple which means inaccuracy.

Applying the new idea to paths: simplifying a path

So, that all was about *approximating a curve*. Then it dawned on me that we could use the same idea to *simplify a path*. Namely, instead of a curve given by some mathematical expression, we can use any curve, for instance a composite Bézier curve (curve consisting of butting Bézier arcs), such as a stroke of a path. Or even something else.

So I dropped what I was doing and started to write a plugin to simplify paths (thinking it would be a quick job...). To make it in any way practical I had to add new twists:

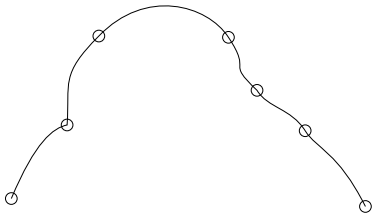
- I wanted to allow the user to set some anchors protected so that the process would not touch them. I did this by subdividing the strokes at such anchors and processing each part separately.

- I wanted to smooth (large) corners that the user did not want to be preserved. For this I wrote routines.

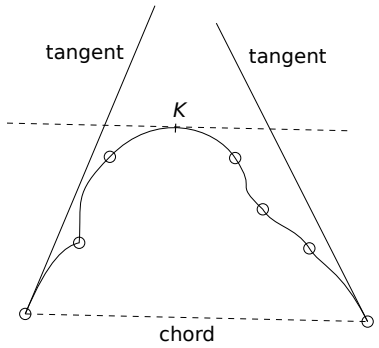
And other minor problems.

How it looks like

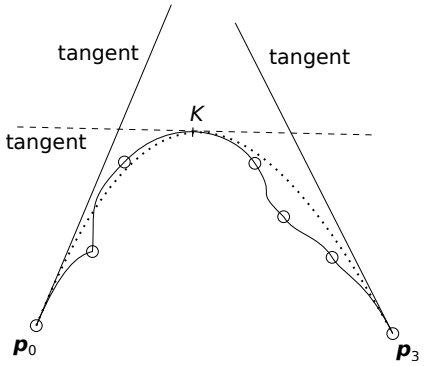
Consider a path (more precisely: a stroke, or a part of a stroke) which we want to simplify:



Find tangents at the end points. Find the point K farthest from the chord. Recall that normally the path then has at K a tangent parallel to the chord.



Given these three tangents (through $\boldsymbol{p}_0$, $\boldsymbol{p}_3$, K), try to find a Bézier arc having the same tangents at the same points. This is where the derived formulas (not shown here) are used to compute handles $\boldsymbol{p}_1$, $\boldsymbol{p}_2$. The dotted arc below might be one such try:



Then measure the error: how much does the computed arc deviate from the original path. If the error is below the allowed limit, accept the approximation. Otherwise do as follows: Split the small top arc in two at K, creating there a new anchor. Then call the algorithm (recursively) for both halfs $\boldsymbol{p}_0K$ and $K\boldsymbol{p}_3$. And so on.

You see that the splitting step creates a new anchor at K. On the other hand, original anchors, except for $\boldsymbol{p}_0$ and $\boldsymbol{p}_3$, are thrown away. Therefore, for the most part the simplified path will have new anchors.