Finding all intersections of paths in MetaPost

Abstract
In this article we will discuss different ways to implement macros to find all intersections of paths in MetaPost. We will first work out some rather simple ideas, providing partial solutions on the macro level. We then show by an example how the `intersectiontimes` macro works, and describe how it was extended in the engine by Hans Hagen.

Introduction

In MetaPost the fundamental way to find intersections of paths is to first find the times of the intersections by running

```
p intersectiontimes q
```

If the paths `p` and `q` intersect, this will return a pair of times `(t1,t2)` such that the path `p` at time `t1` intersect the path `q` at time `t2`. Often, *but not always*, this will return the first intersection point of the paths `p` and `q`, in the sense that `t1` will be the smallest time along `p` for which the paths intersect. The uncertainty comes from the implementation, that relies on a bisection algorithm for finding intersection points of Bézier curves. The algorithm is inherited from MetaFont, and described in detail in [Knu86], but we will also illustrate it later.

In Figure 1 we have drawn a path `p` in blue. Along this path we have placed ten different circles, each playing the rôle of `q`. We run `p intersectiontimes q` and draw with a green dot the point that correspond to the time returned. For the first four circles and for the tenth circle, `intersectiontimes` returns the second intersection point. For the fifth to ninth circle it returns the first intersection point.[1]

```
\startMPcode[instance=doublefun]
numeric n ; n:=10 ;
path p,q ;

p = (0,0) .. (6cm,4cm){dir 20} .. {up}(6cm,0) ;
drawarrow p withcolor darkblue ;

for i = 1 upto n :
  q := fullcircle scaled 30bp shifted point (i/(n+1)) along p ;
  drawarrow q withcolor darkred ;
  tone := xpart (p intersectiontimes q) ;
  drawdot point tone of p withpen pencircle scaled 6bp
    withcolor darkgreen ;
endfor ;
\stopMPcode
```

1. We will often work with pairs of paths `p` and `q`, and since the order will be important, we will always draw the first one (`p`) in blue and the second one (`q`) in red. Moreover, green dots will be used for the built-in macros (or ones written by others). We use orange dots to show our constructed points.

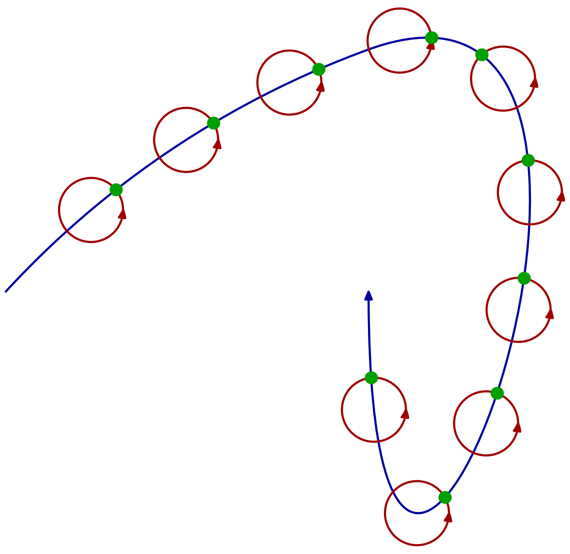


Figure 1.

This is sometimes inconvenient. For example, if we want to cut one path after it intersects another path for the first time (usually done with `cutafter`), or if we want to find all intersection points of two paths (how to loop?), and it would be desirable to have a more reliable algorithm for finding the first intersection point (or all intersection points) of two paths. We are not alone in thinking so[2].

The macro `cutbefore` is used as `p cutbefore q`; it is supposed to return the part of the path `p` that remains after the *first* intersection of `p` and `q`. It often does, but in the MetaPost manual [Hob20] we can read that `p cutbefore q` is equivalent to

```
subpath (xpart(p intersectiontimes q), length p) of q
```

As `intersectiontimes` is used, we will not always get the first intersection, and therefore the user must be careful to see that the result is the intended.

```
\startMPcode[instance=doublefun]
path p, q;
p := (0,0) -- (5cm,0) ;
q := fullcircle rotated -20 scaled 2cm xshifted 2.5cm ;
pair b ; b := (p intersectiontimes q) ;
drawarrow p withcolor darkblue ;
drawarrow q withcolor darkred ;
drawdot point xpart b of p withpen pencircle scaled 6bp
  withcolor darkgreen ;
\stopMPcode
```

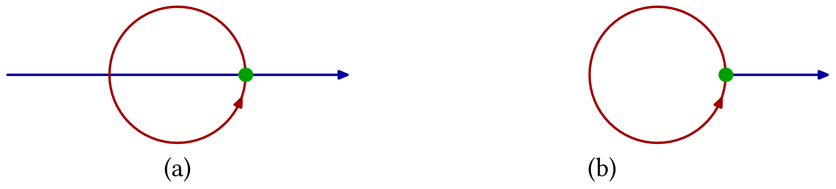


Figure 2. (a) For this pair of curves `intersectiontimes` returns the times for the second intersection point. (b) As a consequence `cutbefore` cuts at the time that `intersectiontimes` returned.

2. See for example the questions https://tex.stackexchange.com/q/79256/52406 and https://tex.stackexchange.com/q/180510/52406.

MetaPost paths are piecewise built with help of Bézier curves, polynomials of degree three. For us it is convenient to think of a path `p` as a parametrized path where the parameter `t`, which we usually call time, runs over a certain interval with endpoints `0` and `length p`. For the path `p` in figure 1 it holds that `length p` equals `2`; the three points used in the definition correspond to time `0`, time `1` and time `2`. We call the parts between the points the *segments* of the path. Each segment corresponds to a unit time interval.

In the search for solutions

At TEX StackExchange, in an answer[3] to the question mentioned before, we find a suggestion on how to find all intersection points of two paths. It is based on a suggestion on the MetaPost mailing list [Hen08].

```
\startMPcode[instance=doublefun]
path p, q, r ;
p := fullcircle xscaled 144 yscaled 72 ;
q := fullcircle xscaled 72 yscaled 144 ;
r := p ;
drawarrow p withcolor darkblue ;
drawarrow q withcolor darkred ;
n := 0 ;
forever :
  r := r cutbefore q ;
  exitif length cuttings = 0 ;
  r := subpath(epsilon, length r) of r ;
  z[n] = point 0 of r ;
  drawdot z[n] withpen pencircle scaled 6bp
    withcolor darkgreen ;
  n := n + 1 ;
endfor;
\stopMPcode
```

As you can see in Figure 3(a) the intersection points are found. In Figure 3(b) we use the same method, but with the paths `p` and `q` changed into

```
p:= (0, 0) -- (5cm, 0) ;
q:= fullcircle rotated -20 scaled 2cm xshifted 2.5cm ;
```

only the right intersection point is found, which is not a surprise to you by now.

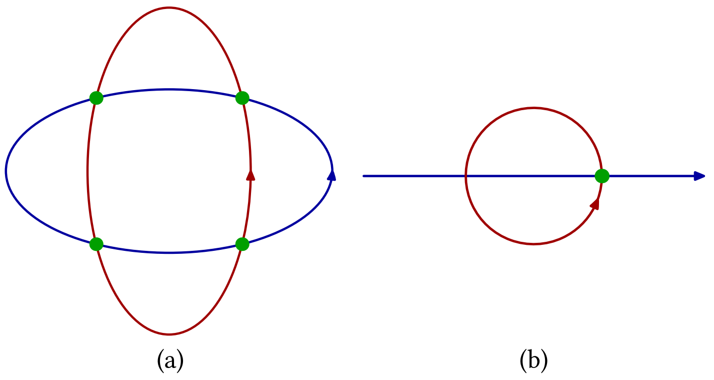


Figure 3.

3. https://tex.stackexchange.com/a/79332/52406

The document [Thu17] contains a lot of beautiful MetaPost graphics to get inspiration from. It contains also a nice discussion on the difficulty of finding all intersection points of two curves (Sections 9.4.1 and 9.4.1), concluding with essentially the same suggestion as the one given above.

To find the first intersection time, a first try

One way to find all intersection points of two paths could be to first have a method to find the first intersection point. We will try to find the first intersection point by repeating the use of `intersectiontimes` on smaller and smaller subpaths.

We illustrate the idea by looking at the line and the circle again. First we use `p intersectiontimes q` to find one (any) intersection point, see Figure 4(a). Then we cut away the part of the path `p` that comes after that intersection. This is shown in Figure 4(b). In fact, we cut just a bit more, to be sure not to arrive at the point we just got. Once that is done, we repeat, see Figure 4(c), where a second intersection is found. In our example, this is the last one, since once we cut again, the paths do not intersect any more, see Figure 4(d). The algorithm returns the time of the last intersection point that was found, in our case the time of the orange dot marked in Figure 4(c).

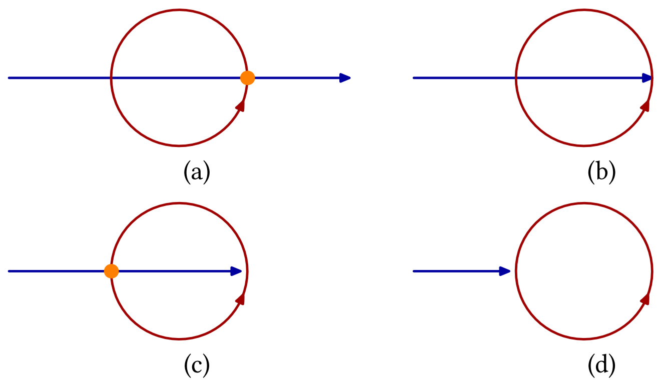


Figure 4.

We implement this algorithm below.

```
\startMPdefinitions{doublefun}
tertiarydef p firstintersectiontimetryone q =
  begingroup;
  save tres, tmpt, tmpp, tmpl ;
  pair tres ; tres := (-1, -1) ;
  pair tmpt ; tmpt := (length p, 0) ;
  path tmpp ; tmpp:=p;
  numeric tmpl ;
  forever :
    tmpt := tmpp intersectiontimes q ;
    exitif xpart tmpt < 0 ;
    tmpl := arclength subpath(0, xpart tmpt) of tmpp ;
    tres := (arctime tmpl of p, ypart tmpt) ;
    tmpp := subpath(0, (xpart tres) - epsilon) of p ;
  endfor ;
  tres
  endgroup
enddef ;
\stopMPdefinitions
```

It might be helpful to make a few comments. The `tmpl` variable might at a first glance seem superfluous. But the `tmpt` variable calculates the time along `tmpp`. It doesn't necessarily hold that this time is the same as the time along the original path `p`. So, we work with lengths instead, since they do not change with parametrizations. We first use `arclength` set `tmpl` to the length of the part of the temporary path `tmpp` before the cut. Then we use `arctime` to calculate the time on the original path `p` that corresponds to this length. This is probably a stupid thing to do numerically, since there are several points where inaccuracies might be introduced, but at least it seems to work for simple examples. When the conversion is done we update the temporary path `tmpp`, and cut away an `epsilon` (that is $1/65536$) of it in the end, not to get stuck in an infinite loop.

We test this definition on the different combinations of `p` and `q` from Figure 1. This is done with the following code[4], and the result is shown in Figure 5.

```
\startMPcode[instance=doublefun]
numeric n, standardfirst, ourfirst ; n := 10 ;
path p,q ; p := (0, 0) .. (6cm, 4cm){dir 20} .. {up}(6cm, 0) ;
drawarrow p withcolor darkblue ;

for i = 1 upto n :
  q := fullcircle scaled 30bp shifted point (i / (n + 1)) along p ;
  drawarrow q withcolor darkred ;
  standardfirst := xpart (p intersectiontimes q) ;
  drawdot point standardfirst of p withpen pencircle scaled 6bp
    withcolor darkgreen ;
  ourfirst := xpart (p firstintersectiontimetryone q) ;
  drawdot point ourfirst of p withpen pencircle scaled 4bp
    withcolor "orange" ;
endfor;
\stopMPcode
```

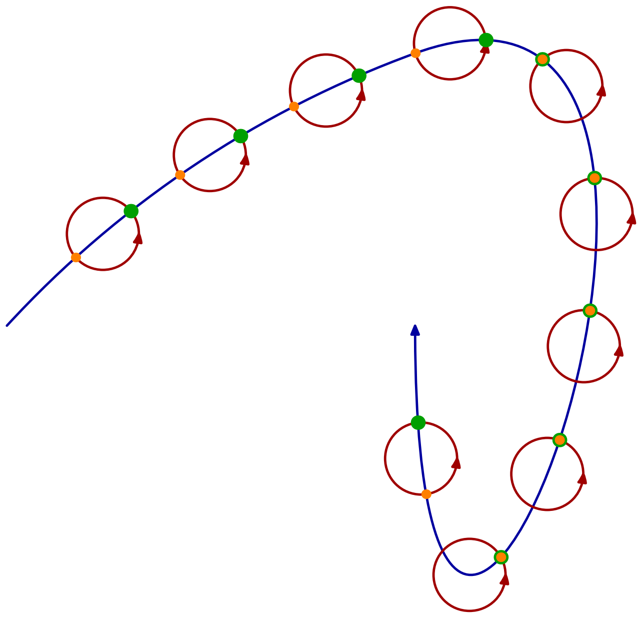


Figure 5.

4. Note that we write `darkred` and `darkblue` but `"orange"`. The orange color is undefined in MetaPost/MetaFun, so we borrow the definition from ConTEXt.

As you can see, the new macro returns the first intersection points in all ten cases. This could have been the happy end, but, as we will see, it is not. Not to be too crestfallen at this point, we show in Figure 6 some examples where the algorithm works.

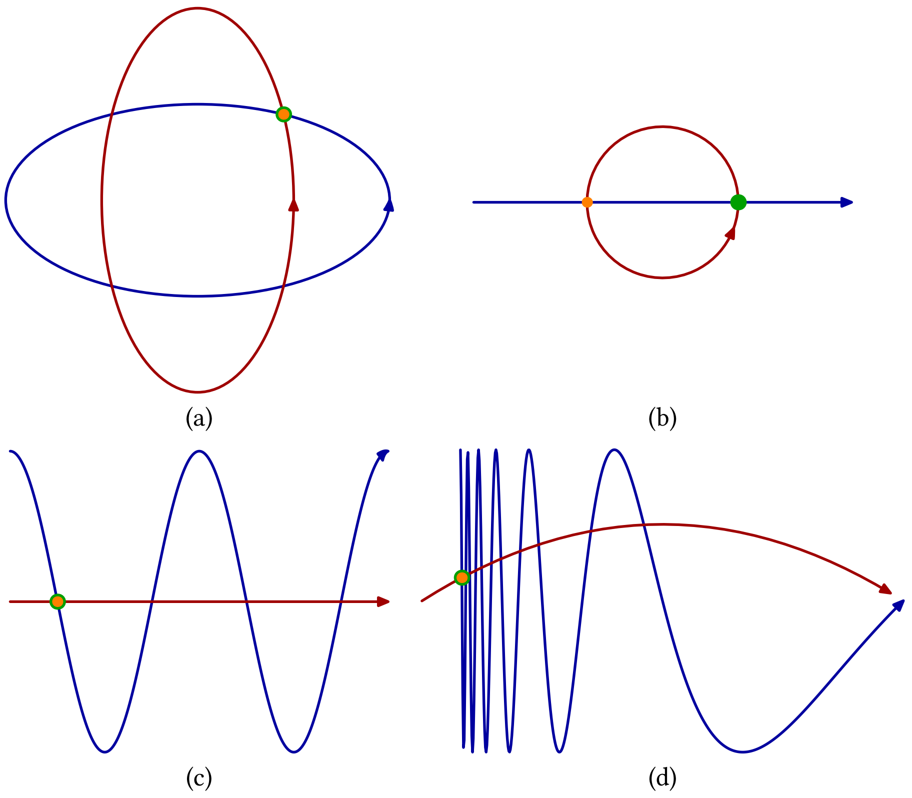


Figure 6.

To find the first intersection time, a second try

If the paths p and q start at the same point we have a problem. Our algorithm gets stuck in an infinite loop. This happens for example for the paths

```
p := (0, 0){dir 45} .. (5cm, 1cm) ;
q := (0, 0) -- (5cm, 0cm) ;
```

The standard solution with p `intersectiontimes` q returns the intersection at time zero, and we have marked that point in Figure 7.

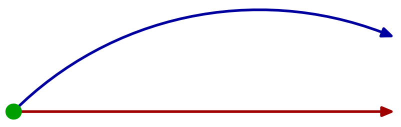


Figure 7.

To overcome this problem we introduce a test to check if we have ended up at the beginning of the path. We introduce a boolean `tzero` that we set to `true` if the intersection time found is less than a certain value, which we have chosen to be `epsilon`.

```
\startMPdefinitions{doublefun}
tertiarydef p firstintersectiontimetrytwo q =
  begingroup ;
  save tres, tmpt, tmpp, tmpl, tzero ;
  pair tres ; tres := (-1, -1) ;
  pair tmpt ; tmpt := (length p, 0) ;
  path tmpp ; tmpp := p ;
  numeric tmpl ;
  boolean tzero ; tzero := false ;
  forever :
    tmpt := tmpp intersectiontimes q ;
    exitif xpart tmpt < 0 ;
    if xpart tmpt < epsilon :
      tzero := true ;
      exitif true ;
    fi
    tmpl := arclength subpath(0, xpart tmpt) of tmpp ;
    tres := (arctime tmpl of p, ypart tmpt) ;
    tmpp := subpath(0, (xpart tres) - epsilon) of p ;
  endfor ;
  if tzero :
    tmpt
  else :
    tres
  fi
  endgroup
enddef ;
\stopMPdefinitions
```

In Figure 8(a) we see that our new approach works well, the first point is found. In Figure 8(b) we have just included an example where the paths intersect at the endpoint of p. No problem!

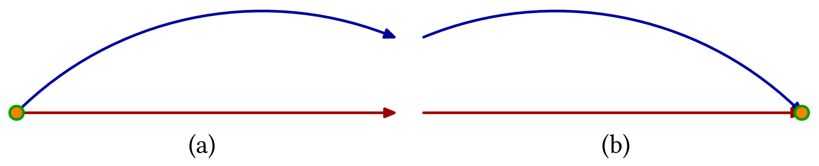


Figure 8.

We will not refine the algorithm for finding the first intersection time further here, but leave only some comments.

The constant epsilon is the *time step* we take from a previously found intersection time. If the path p consists of many points (this is the case for the case of the $\sin(1/x)$ path above, where very small sampling step is chosen) then this means that two points in the path can lie very close to each other. The time step between them is still one, so an epsilon time step away along the path might result in indistinguishable points, and that might result in an infinite loop. One way to solve such an issue could be to add a new variable to the macro that replaces epsilon. That would require more from the user; for example it would be necessary to have the sampling step in function graphs in mind while working with intersections.

Instead of removing an epsilon time part of the path, one could convert to lengths and remove an epsilon length of the path. That would lead to a potential lost of accuracy in the calculations, but it could at the same time be easier for the user to set a threshold in terms of length instead of time.

To find all intersection times, a first try

With the time for the first intersection at hand, it is in principle straightforward to find all intersection times. We illustrate with the line and the circle again. In Figure 9(a) we have marked the first intersection point. This time we cut away the part of the path p *before* the intersection point. This is done in Figure 9(b). Again, we need to cut a bit more, not to end up with an infinite loop. So, in Figure 9(c) we have cut just a bit more, and the next intersection is found. Finally, in Figure 9(d) we have cut away so much that the paths do not intersect anymore. We have found all intersections.

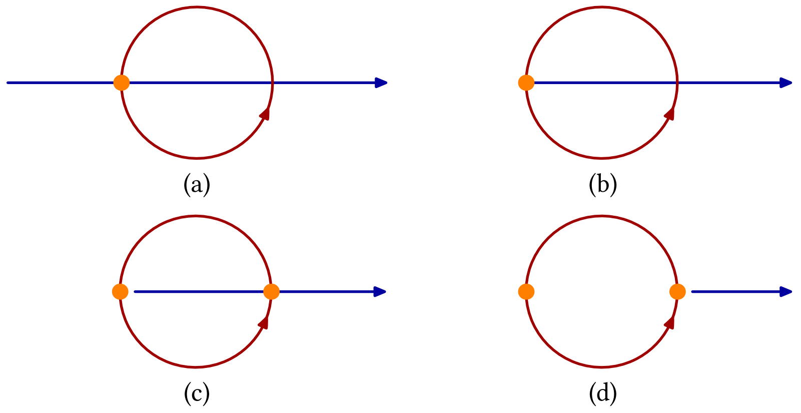


Figure 9.

We try with the following algorithm.

```
\startMPdefinitions{doublefun}
tertiarydef p allintersectiontimestryone q =
  begingroup ;
  save tmpt, tmpp, tmpptmp, tmpl, it, n ;
  pair tmpt ;
  path tmpp, tmpptmp ; tmpp := p ;
  numeric tmpl ;
  pair it[] ;
  numeric n ; n := 0 ;
  forever :
    tmpt := tmpp firstintersectiontimetrytwo q ;
    exitif xpart tmpt < 0 ;
    tmpptmp := subpath(xpart tmpt, length tmpp) of tmpp ;
    tmpl := arclength p - arclength tmpptmp ;
    it[incr n] := (arctime tmpl of p, ypart tmpt) ;
    tmpp := subpath(xpart tmpt + eps, length tmpp) of tmpp ;
  endfor ;
  topath(it, --) % it[1] -- it[2] -- ... -- it[n]
  endgroup
enddef ;
\stopMPdefinitions
```

We first test it on the examples from Figure 6. The code for the ellipses is shown below, the other are similar. As you can see in Figure 10 it works in all cases.

```
\startMPcode[instance=doublefun]
path p ; p := fullcircle xscaled 144 yscaled 72 ;
path q ; q := fullcircle xscaled 72 yscaled 144 ;
```

```
drawarrow p withcolor darkblue ;
drawarrow q withcolor darkred ;

path b ; b := p allintersectiontimestryone q ;

for i = 0 upto length b :
  drawdot point xpart (point i of b) of p
    withpen pencircle scaled 6bp
    withcolor "orange" ;
endfor ;
\stopMPcode
```

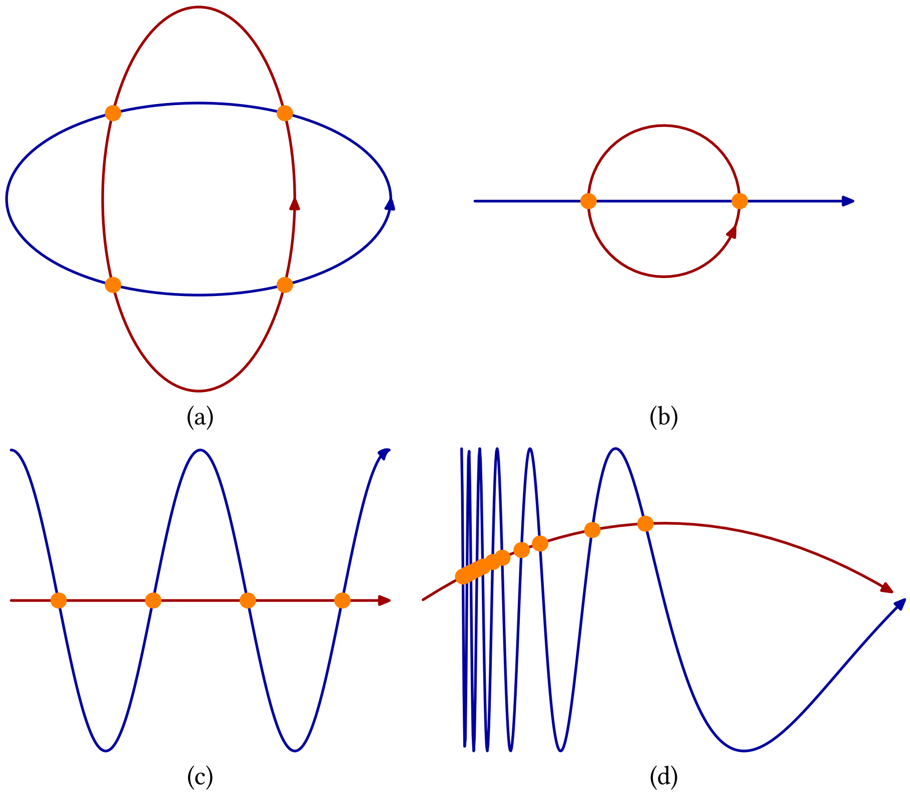


Figure 10.

We also try it on the paths from Figure 5, the result can be seen in Figure 11. Again we get all intersections.

The name of this section suggests that this implementation has problems, and indeed it has. This time it is not the first point of the path p that is the problem, but the last.

To find all intersection times, a second try

The macro allintersectiontimestryone works fine for the paths Figure 8(a) that intersect at the first point of p, but it hangs with an infinite loop for the paths in Figure 8(b). The reason is that the paths intersect at the endpoint of p. We must, similarly as we did when we found the first intersection point, check if we are at the end of the path. We rewrite the macro like this.

```
\startMPdefinitions{doublefun}
tertiarydef p allintersectiontimes q =
  begingroup ;
  save tmpt, tmpp, tmpptmp, tmpl, tmpL, it, n, a, cuttime ;
  pair tmpt, it[] ;
  path tmpp, tmpptmp ;
  numeric tmpl, tmpL, n ; n := 0 ;
  tmpp := p ;
  tmpt := tmpp firstintersectiontimetrytwo q ;
  forever:
    exitif xpart tmpt < 0 ;
    exitif length(tmpp) = 0 ;
    tmpptmp := subpath(xpart tmpt, length tmpp) of tmpp ;
    tmpl := arclength p - arclength tmpptmp ;
    it[incr n] := (arctime tmpl of p, ypart tmpt) ;
    tmpL := arclength(subpath(0, xpart tmpt) of tmpp) + eps ;
    if tmpL > arclength tmpp :
      exitif true ;
    else :
      cuttime := arctime (tmpL) of tmpp ;
      tmpp := subpath(cuttime, length tmpp) of tmpp ;
    fi ;
    tmpt := tmpp firstintersectiontimetrytwo q ;
  endfor ;
  topath(it, --)
  endgroup
enddef ;
\stopMPdefinitions
```

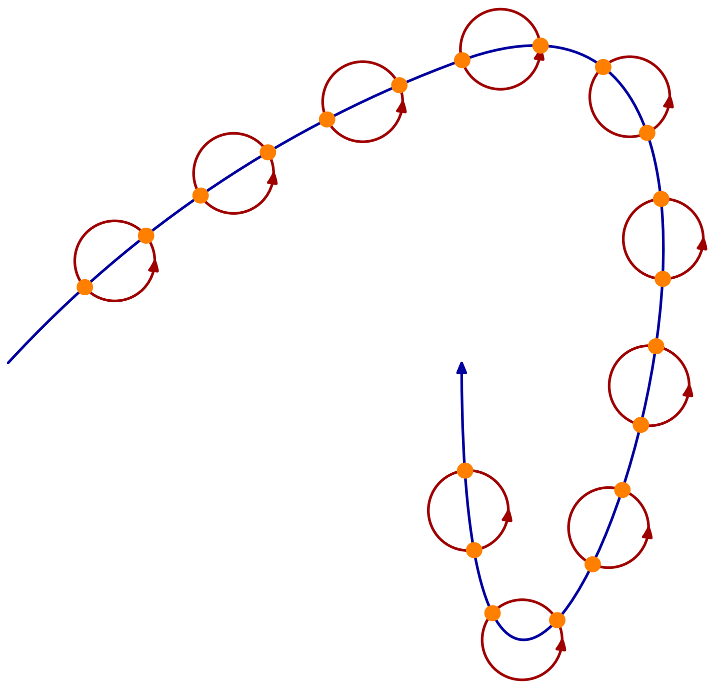


Figure 11.

Here we have introduced `tmpL` to check if we are at the endpoint of the path `p`. As we see in Figure 12 we now catch intersections both at the beginning and at the end of the path `p`.

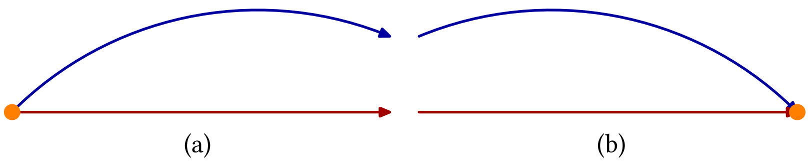


Figure 12.

We could continue here with refinements. We could for example condition on if `p` is a loop or not, we could tune the size of the steps[5] with which we jump ahead after finding an intersection point, and so on. But our final way of solving the problem will be different, so we stop here. You have a macro that works well in most cases.

Returning to the `intersectiontimes` algorithm

I was discussing the problem of finding all intersection points with Hans Hagen, and he had a slightly different idea than the one we have described above. Instead of first iterating to get the first intersection point and then to cut and iterate, Hans wanted to tweak the existing `intersectiontimes` algorithm in order to find all intersection times. Let us first discuss the `intersectiontimes` algorithm in more detail.

We recall that a path in metapost can be thought of as an array[6] of points and control points that together describe the segments that build the path. In Figure 13(a) we have drawn a path with three segments and labeled the four points that define it. Each segment correspond to a unit time interval, and it can mathematically be described by a Bézier curve. In Figure 13(b) we have emphasized the segment between time 2 and time 3, that is `subpath (2,3) of p`. We have also drawn the two control points and control lines of this segment.

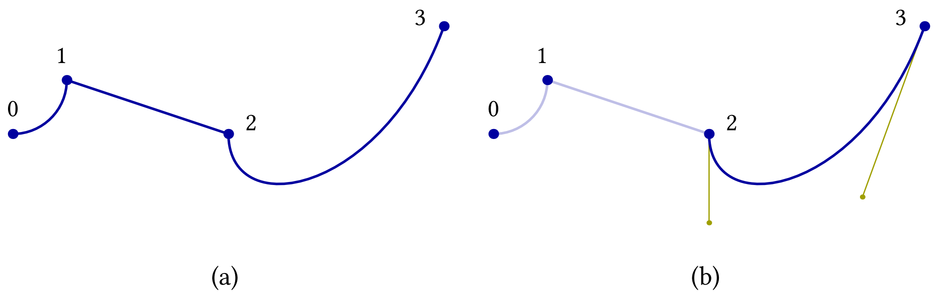


Figure 13.

The `intersectiontimes` macro iterates as mentioned over the different segments of the paths. This reduces the problem of finding all intersection times into the problem of finding the intersection times for every pair of segments. This is done via bisection of the time intervals, working with certain extended boundingboxes (see below). If the parts[7] `pX` of `p` and `qY` of `q` have extended boundingboxes that intersect, they are

5. The `eps` (not `epsilon`) is not a mistake. It will make our examples work. This sensitivity suggests that the method we use is non-optimal.
6. It is in fact a circular linked list. These are more dynamic in memory, and fits well with the use of control points when paths are cycled.
7. Here you can think of `X` and `Y` as a given sequence of zeros and ones. We add `0` for the subpath corresponding to smaller time values, and `1` for larger.

halved (with respect to time) into `pX0`, `pX1`, `qY0` and `qY1`, and the algorithm works on the new parts in the following order:

1. `pX0` is tested against `qY0`,
2. `pX0` is tested against `qY1`,
3. `pX1` is tested against `qY0`,
4. `pX1` is tested against `qY1`.

We give an example to show how the `intersectiontimes` macro works on a pair of paths consisting of two points each (that is of time length 1, or, if we want, with one segment). We have drawn `p` and `q` in Figure 14; the paths intersect at three points. We follow the first few steps of the algorithm in Figures 15–30.

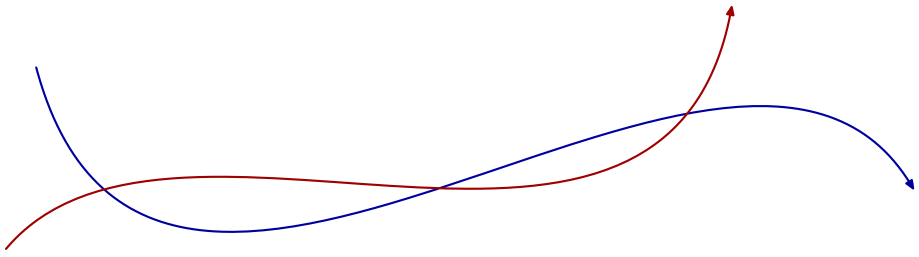


Figure 14.

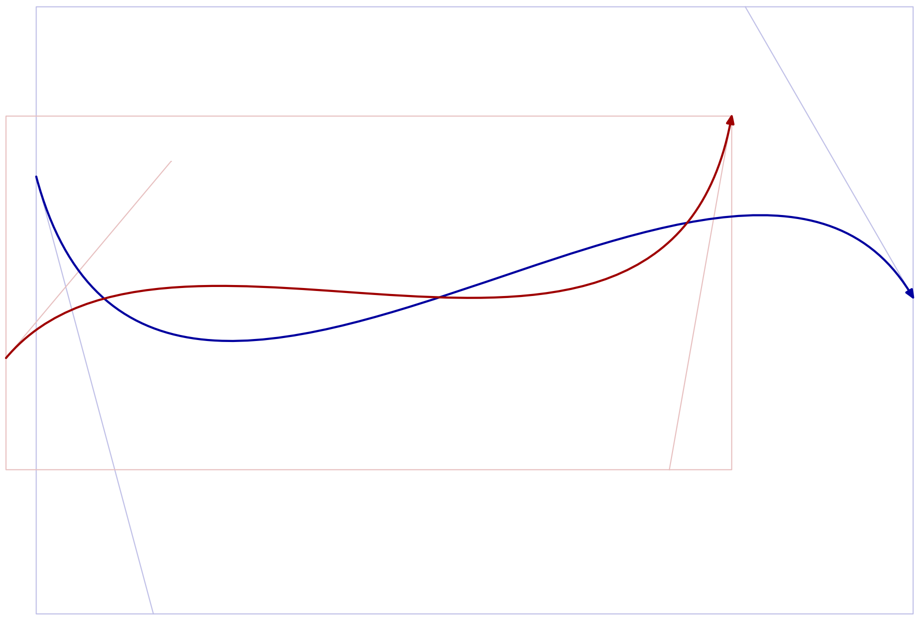


Figure 15. The starting point. For each path we draw its *extended boundingbox*, that is the smallest axis-parallel rectangle that includes both the points and the control points of the path. The extended boundingbox always contains its path, so if the two extended boundinboxes don't intersect[8] then the paths cannot intersect. Here the rectangles intersect. This does not guarantee that the paths intersect; it only means that we should continue with the next step.

8. We say here that two rectangles intersect if they have points (interior, or on the rectangle) in common

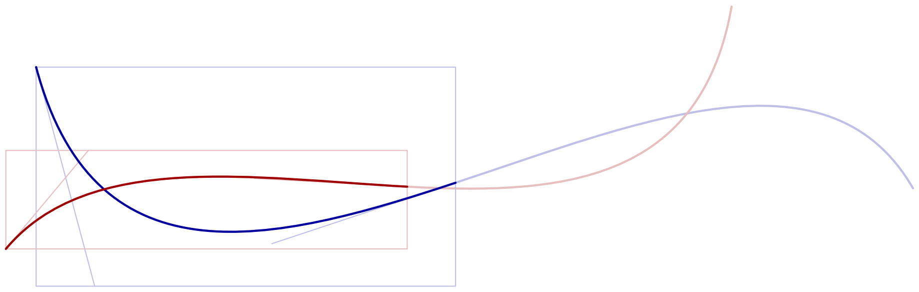


Figure 16. We divide p and q into two pieces at time $1/2$, and call the new paths p0, p1, q0 and q1. The darkblue path is p0 and the darkred one is q1. Since their extended boundingboxes intersect again, we continue with the division.

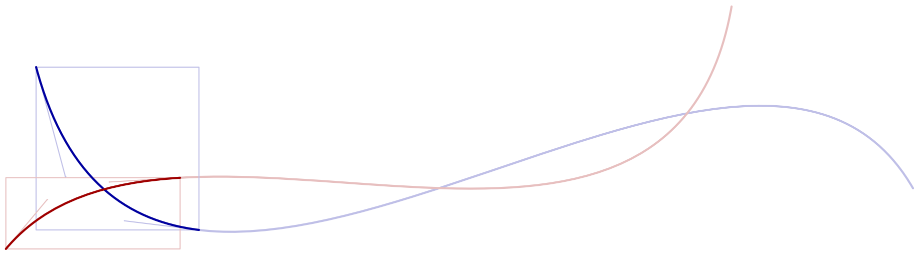


Figure 17. We divide p0 in the middle into p00 and p01. We also divide q0 into two paths q00 and q11. Here we have drawn p00 and q00. Again, we notice that their extended boundingboxes intersect.

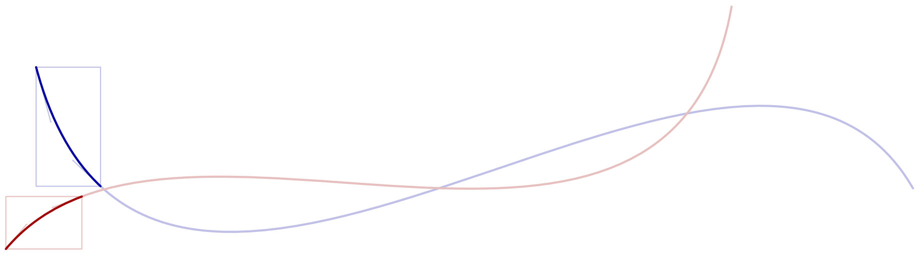


Figure 18. We have divided p00 and q00 in halves. Here we have drawn the paths p000 and q000, and their extended boundingboxes. We note that the boxes do not intersect.

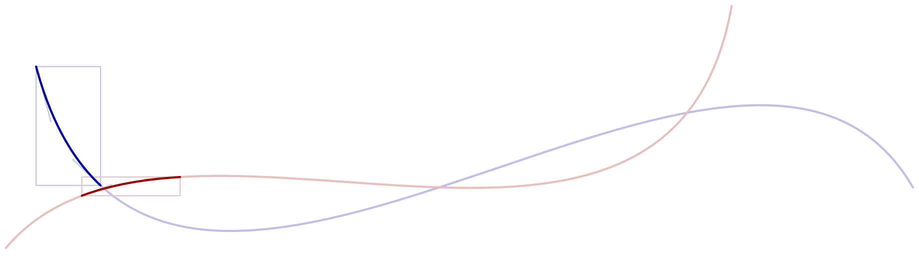


Figure 19. The previous extended boundingboxes did not intersect. We change one of the subpaths, and since we want to have the intersection point as early as possible on the first path p we try first to change the subpath of q. We see that the extended boundingboxes of p000 and q001 do intersect.

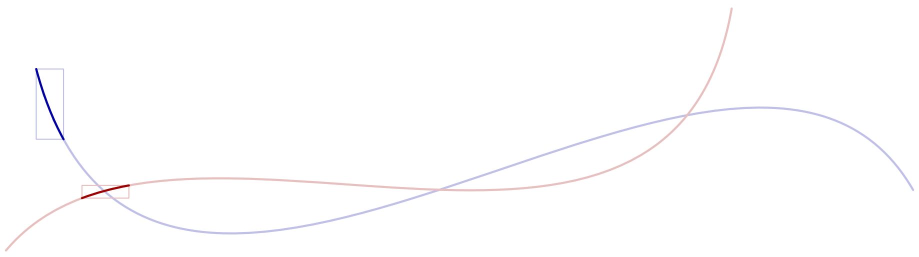


Figure 20. The extended boundingboxes of p`0000` and q`0010` do not intersect.

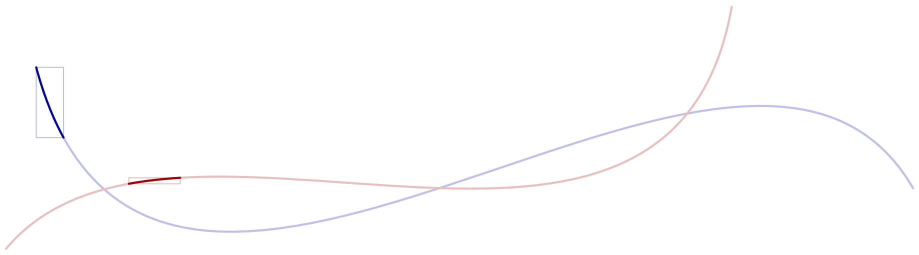


Figure 21. The extended boundingboxes of p`0000` and q`0011` do not intersect.

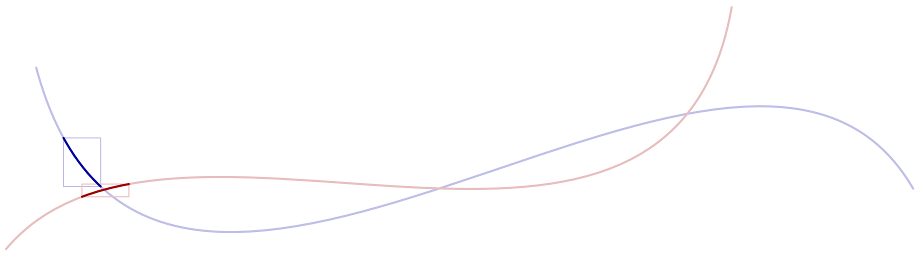


Figure 22. The extended boundingboxes of p`0001` and q`0010` do intersect.

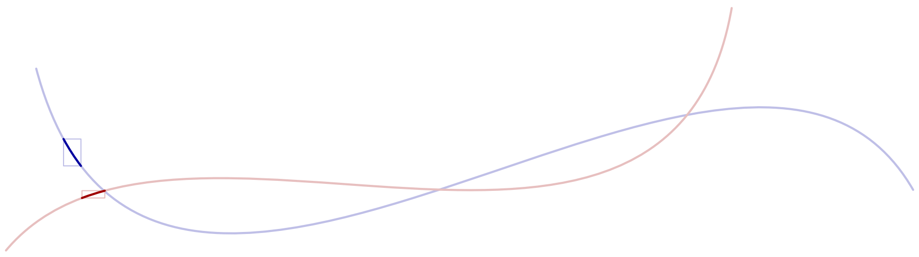


Figure 23. The extended boundingboxes of p`00010` and q`00100` do not intersect.

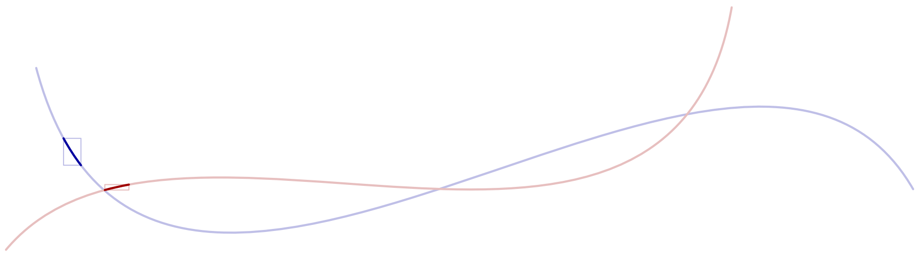


Figure 24. The extended boundingboxes of p`00010` and q`00101` do not intersect.

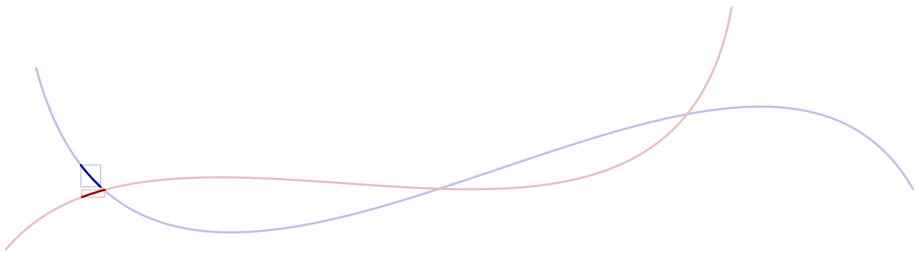


Figure 25. The extended boundingboxes of p`00011` and q`00100` do not intersect.

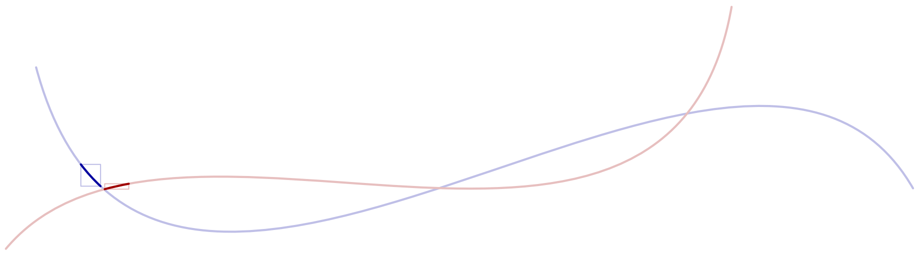


Figure 26. The extended boundingboxes of p`00011` and q`00101` do not intersect. This means that we did enter a dead-end.

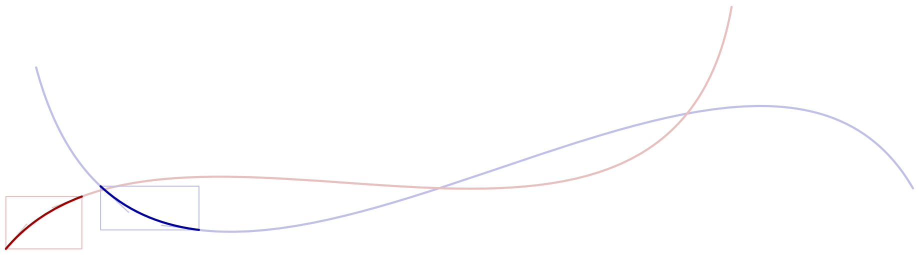


Figure 27. The dead-end pushed us back here. The extended boundingboxes of p`001` and q`000` do not intersect.

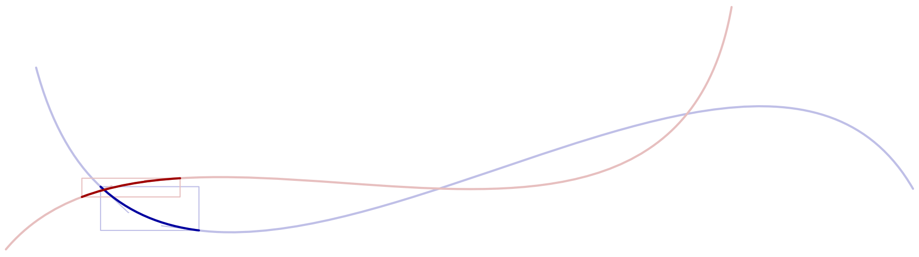


Figure 28. The extended boundingboxes of p`001` and q`001` do intersect.

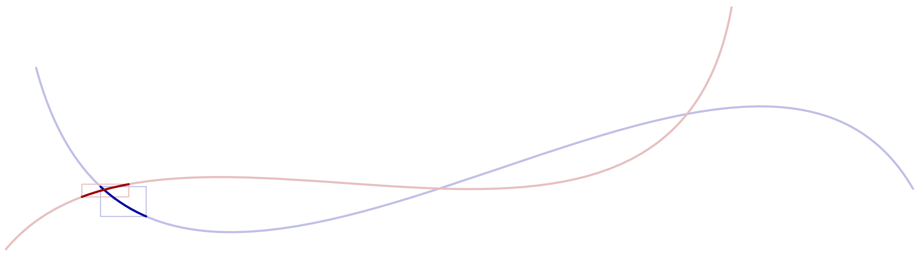


Figure 29. The extended boundingboxes of p`0010` and q`0010` also intersect.

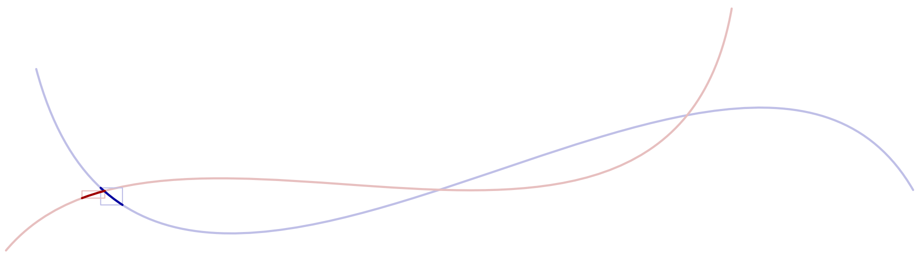


Figure 30. The extended boundingboxes of p00100 and q00100 also intersect.

We could in principle continue this iteration as long as it is numerically meaningful, but we stop here. In Figure 30 we note that the small subpaths actually do intersect. This happens along the path p00100, which means that the time t along p satisfies the inequality

$$\frac{4}{32} = 0.00100_2 \le t \le 0.00101_2 = \frac{5}{32}.$$

Hence, if we set t to be the mean value

$$t = \frac{1}{2}\left(\frac{4}{32} + \frac{5}{32}\right) = \frac{9}{64},$$

then the error (in time!) we make is bounded by

$$\frac{1}{2}\left(\frac{5}{32} - \frac{4}{32}\right) = \frac{1}{64}.$$

In practice the `intersectiontimes` macro has some predefined tolerance, and checks if the size of the extended boundingboxes are smaller than that tolerance. Once the sizes are smaller than the tolerance, it considers the paths to intersect, exits the loop and returns a pair with the times along the paths p and q. If it does not find an intersection, it returns the pair `(-1,-1)`.

The final(?) solution

We saw in the previous section how p `intersectiontimes` q works with bisections to find one intersection of p and q. We have seen that often, but not always, it returns the first intersection (measured in time along p).

Hans Hagen realised that instead of exiting the loop when an intersection is found, one could try again with the same pair segments, but this time neglecting the intersection time that was found. In the code in the engine version the crossing is located with a dedicated function that communicates via variables that are global to the instance.

One could perhaps argue that it would be better to cut and run on the different subpaths, or to just continue directly when the intersection was found. But that would introduce other complications and inaccuracies.

Because we suffer from the same issues as the macro approach we described above, there are three extra tricks used: we have an extra counter that makes sure that when we have some deadlock we can get out of it (like deadcycles in TeX). We also need to ignore duplicates that we get because of the small step and inaccuracies; for that we need to trim 8 bits resolution in order be on the safe side. The test examples of the macro variant was instrumental here. Finally we explicitly need to check the end points because otherwise we miss them, which again sounds familiar from the macro variant.

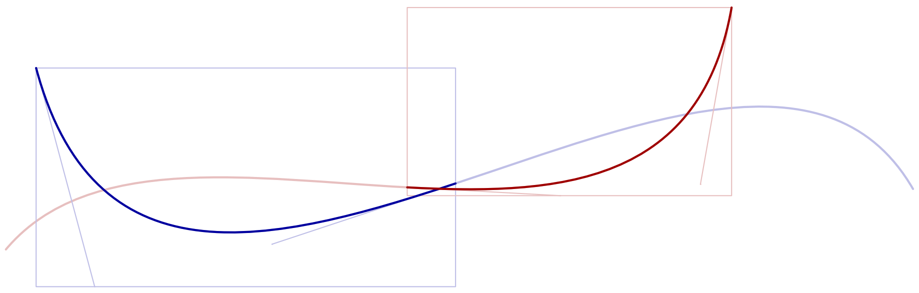


Figure 31. The extended boundingboxes of p0 and q1 do intersect.

In the step-by-step example we have shown above, we only look at the first halves of p and q. All subpaths are followed by a zero, p0X and q0Y. This is what we get in the first run. If we would have continued, we would have got sufficiently small rectangles that meet, and the first intersection time. In the next run, we first end up at the same intersection time, but reject it, and then, eventually, we would end up in the second half of q. The first pair of rectangles will be the ones given in Figure 31. These rectangles intersect, so the loop will continue; in fact it will find the second intersection point here. That would be run two on the segments.

In the third iteration, we will end up at, and skip, the first two intersections. We will then eventually continue by comparing the second half of p with the first half of q. This is seen in Figure 32; the extended bounding boxes do not intersect, so we get no intersection.

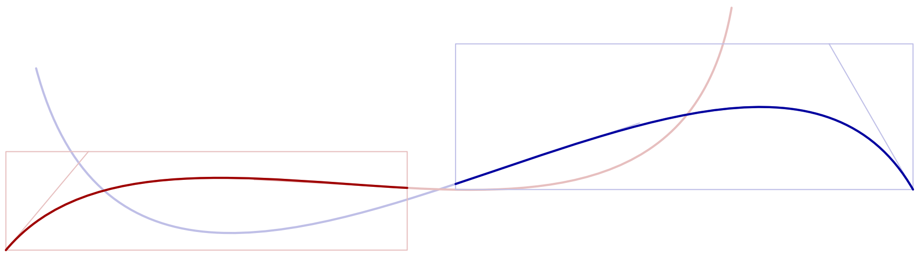


Figure 32. The extended boundingboxes of p1 and q0 do not intersect.

The third iteration continues, and we finally end up in the second half of p and the second half of q, see Figure 33. These intersect, and we will find the third intersection point.

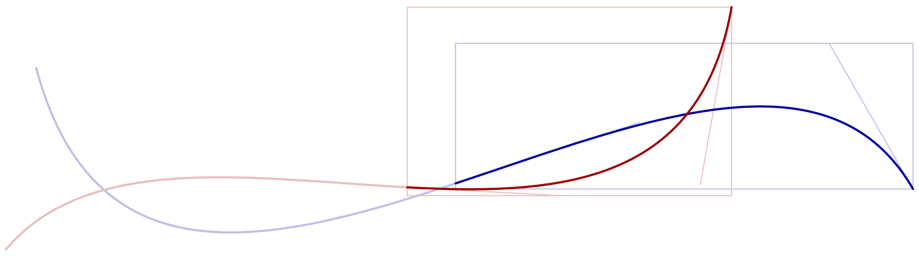


Figure 33. The extended boundingboxes of p1 and q1 do intersect.

When the third intersection point is found, we start the fourth one. This time, we will meet the previous three intersections, and then we will not find any more. The loop is complete. In Figure 34 we have marked the intersections found by the algorithm.

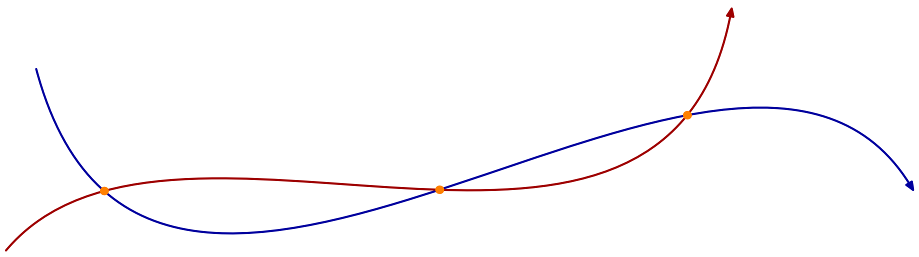


Figure 34.

The new macros

To sum up, we now proudly present a set of new macros. First of all

```
p intersectiontimeslist q
```

This one takes two paths `p` and `q` and returns a path with the intersection times, where the first coordinate of each point is the time along `p` and the second the time along `q`. Then we have

```
p sortedintersectiontimes q
```

that also returns a path with the intersection times of the paths `p` and `q`, sorted in increasing order with respect to the times of `p`.

To find all intersection points, there is

```
p intersectionpath q
```

that returns a path consisting of the points where `p` and `q` intersect, sorted so that the points appear in increasing order with respect to time along `p`. In fact, this macro first uses `intersectiontimes` to find the times, and then for each pair `t` of intersection times it returns the middle point of the paths, that is `0.5[xpart t of p,ypart t of q]`. This is the classical approach of `intersectionpoint`. It might not always be what we want, since the points might not be on any of the two paths. To use the points of `p` or `q` one can instead use

```
p firstintersectionpath q
```

or

```
p secondintersectionpath q
```

Since the macros `cutbefore` and `cutafter` are relying on `intersectionpoints` we have also introduced

```
p cutbeforefirst q
```

that returns the part of `p` that remains when the part before the first intersection point with `q` is cut away, and

```
p cutafterfirst q
```

that returns the part of `p` that remains when the part after the first intersection point with `q` is cut away. Similarily, there are

```
p cutbeforelast q
```

and

```
p cutafterlast q
```

that work on the last intersection (along the first path `p`) of `p` and `q`.

Other tools

It is also possible to find all intersection points by using some other common drawing tools. In Figure 35 we show the intersectionpoints of the curves $y = 2\sin(2\pi x)$ and $y = 1/(1+2x)$ for $0 \leq x \leq 5$. We say something about each tool below, and provide the code used in each case.

Asymptote

Asymptote is a relative to MetaPost, and it seems that its macro `intersection` uses a similar approach as `intersectiontimes`, inherited from MetaFont. We find the following in Section 6.2 on Paths and guides of the manual [HBP22]:

> "If `p` and `q` have at least one intersection point, return a real array of length 2 containing the times representing the respective path times along `p` and `q`, in the sense of `point(path, real)`, for one such intersection point (as chosen by the algorithm described on page 137 of The MetaFont book). The computations are performed to the absolute error specified by `fuzz`, or if `fuzz < 0`, to machine precision. If the paths do not intersect, return a real array of length 0."

The `intersections` (plural!) macro is used to find all intersection points of two paths. It is not mentioned in the just cited document how it obtains all the intersection points. The code below was placed in a file, and then we ran `asy` on it.

```
unitsize(1cm) ;
import graph ;

real f(real x) { return 2*(sin(2*pi*x)+1) ; }
path A = graph(f,0,5,n=200) ;
draw(A, arrow = Arrow(size = 5), rgb(0, 0, 0.5) + linewidth(1bp)) ;

real g(real x) { return 4/(1+2*x) ; }
path B = graph(g,0,5,n=200) ;
draw(B, rgb(0.5, 0, 0) + linewidth(1bp)) ;

pair vert[] = intersectionpoints(A, B) ;
for(int k = 0 ; k <= vert.length-1; ++k){
  dot(vert[k], orange + linewidth(6bp)) ; }
```

MetaPost/MetaFun

We use the new macro `intersectionpath` to find all intersections between two paths.

```
\startMPcode[instance=doublefun]
path A, B, C ;
A := function(1, "x", "2*sin(2*pi*x) + 2" ,0 ,5 ,1/40) scaled(1cm) ;
B := function(1, "x", "4/(1 + 2*x)" , 0, 5, 1/40) scaled(1cm) ;
C := (A intersectionpath B) ;
drawarrow A withcolor darkblue ;
draw B withcolor darkred ;
drawpoints C withpen pencircle scaled 6bp withcolor "orange" ;
\stopMPcode
```

Pstricks

For pstricks one can use the `pst-intersect` package [Ber14]. It is not clear from the manual how it finds the intersections, but skimming the code, there seems to be some kind of bisection going on, with the clipping of paths (my brain is not constructed to read PostScript code). It is mentioned in the manual that it borrowed the Graham Scal algorithm to calculate the convex hull of a set of points from Bill Casselman's wonderful book *Mathematical Illustrations* (kindly made available online [Cas05]). The implementation in `pst-intersect` can also handle higher order Bézier curves, and perhaps it is for these that the convex hull algorithm is needed. The LaTeX packages `multi-do`, `pst-intersect` and `xcolor` were loaded in the example below.

```
\begin{pspicture}(5,4.4)
\pssavepath[linecolor=blue!50!black,linewidth=1bp]{A}{
  \psplot[plotpoints=200,arrows=->]{0}{5}{x 360 mul sin 1 add 2 mul} }
\pssavepath[linecolor=red!50!black,linewidth=1bp]{B}{
  \psplot[plotpoints=50]{0}{5}{4 2 x mul 1 add div} }
\psintersect[linecolor=orange,showpoints,linewidth=2bp]{A}{B}
\end{pspicture}
```

Tikz

Tikz has in the `intersections` library support for finding all intersections of two arbitrary paths. It is not clear from the manual [Tan22] how it works, but in Section 13.3.2 we can read

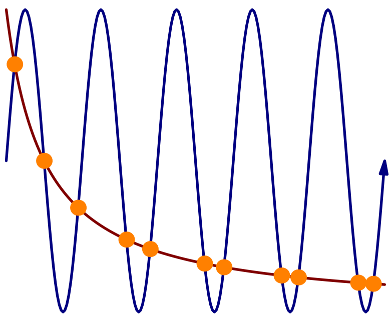
> "This library enables the calculation of intersections of two arbitrary paths. However, due to the low accuracy of TeX, the paths should not be 'too complicated'. In particular, you should not try to intersect paths consisting of lots of very small segments such as plots or decorated paths."

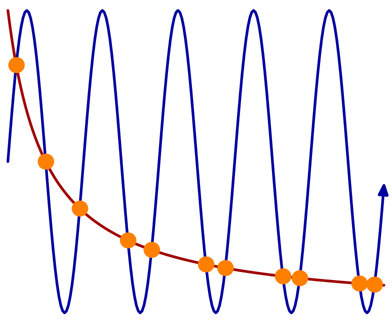
In the example below we have loaded the `tikz` package, the `intersections` library and also defined two colors

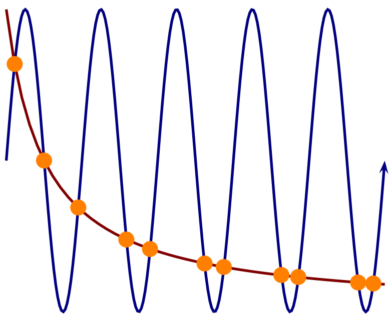
```
\definecolor{darkblue}{rgb}{0, 0, 0.5}
\definecolor{darkred}{rgb}{0.5, 0, 0}
```

Then we used this code.

```
\begin{tikzpicture}
\draw[name path=A, variable=\x, domain=0:5, samples=200,
      line width=1bp, smooth, color=darkblue, ->]
    plot (\x, {2*sin(2*pi*\x r) + 2});
\draw[name path=B, variable=\x, domain=0:5, samples=200,
      line width=1bp, smooth, color=darkred]
    plot (\x, {4/(1+2*\x)});
\fill[name intersections={of=A and B, sort by=line,
      name=i, total=\t}, color=orange]
  \foreach \s in {1,...,\t}{(i-\s) circle (3bp)};
\end{tikzpicture}
```


(a) asymptote


(b) metapost


(c) pstricks

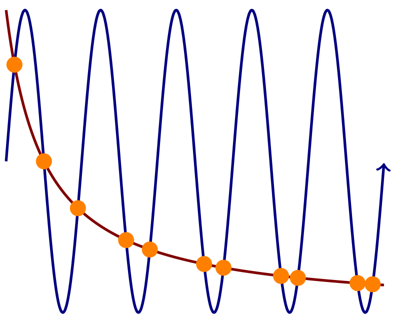

(d) tikz

Figure 35.

Final words

As we have seen, to find all intersections of two paths is not a trivial task, but now we MetaPost users have reliable engine macros that work. Even with our new macros, however, things can go wrong. Some paths are simply tricky to work with, and sometimes some manual tweaking is required. Let us give one such example.

We try to find the intersections of $y = 4\cos(x) + \cos(2x)$ and $y = 5\cos(x)$, for $0 \le x \le 8\pi$. The problem with these curves is that they coincide to order two at the maximas, located at $x = 2\pi k$ for $k \in \{0, 1, 2, 3, 4\}$. This means that the curves are not really crossing there. One could probably find good arguments both for and against that these points should be returned by the algorithm. As the situation is, it returns some, and it depends on how many points we use when constructing the paths. In the code below we use a step size of 0.05 for x. As can be seen in Figure 36, some points are missing. If we change the step size to 0.01 we will see that all the points except the last one is there.

```
\startMPcode
path p, q, b ;
p := function(2, "x", "4*cos(x)+cos(2*x)", epsed(0), epsed(8*pi), 0.05)
  scaled 15 ;
q := function(2, "x", "5*cos(x)", epsed(0), epsed(8*pi), 0.05) scaled 15;
b := (p intersectionpath q) ;

drawarrow p withcolor darkblue;
drawarrow q withcolor darkred;
drawpoints b withcolor "orange" ;
\stopMPcode
```

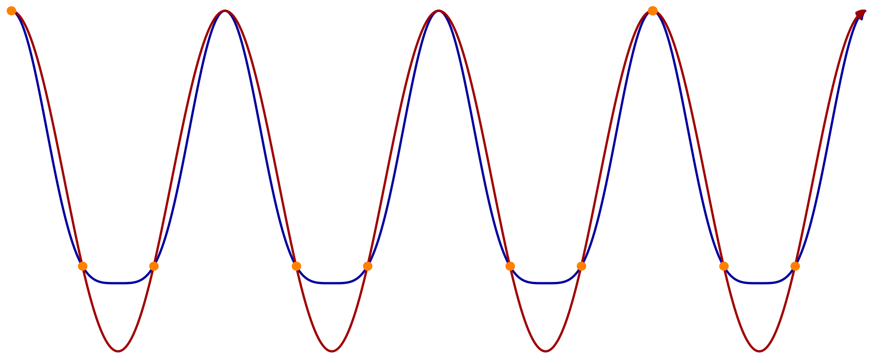


Figure 36.

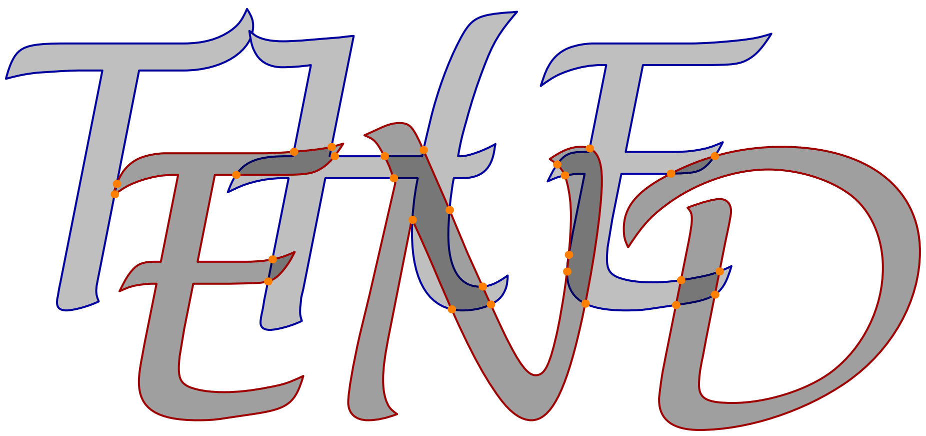
It would feel wrong to end this article, that is written with a big smile in the face, with a negative example. We give instead an example where all is going well, and why not use the nice feature to extract outlines of characters?

```
\startMPcode
picture p ; p := lmt_outline [ text = "THE" ] scaled 15 ;
picture q ; q := lmt_outline [ text = "END" ] scaled 15 ;
q := q shifted (60, -50) ;

for pp within p :
  fill pathpart pp withcolor 0.75white ;
  draw pathpart pp withpen pencircle scaled 1bp withcolor darkblue ;
endfor ;

for qq within q :
  fill pathpart qq withcolor 0.25white withtransparency (2, 0.5) ;
  draw pathpart qq withpen pencircle scaled 1bp withcolor darkred ;
endfor ;

for pp within p :
  for qq within q :
    path r ; r := (pathpart pp) intersectionpath (pathpart qq) ;
    if known r :
      drawpoints r withpen pencircle scaled 4bp withcolor "orange" ;
    fi ;
  endfor ;
endfor ;
\stopMPcode
```



Acknowledgements

Neither the new engine macros nor this article would have existed if it not were for Hans Hagen, and his open mind to new ideas. Thank you!

References

[HBP22] A. Hammerlindl, J. Bowman, and T. Prince, *Asymptote*, https://asymptote.sourceforge.io/doc/index.html (2022). (version: 2.80-35)

[Cas05] B. Casselman, http://www.math.ubc.ca/ cass/graphics/text/www/ (2005).

[Hen08] T. Henderson, https://tug.org/pipermail/metapost/2008-October/001467.html (2008). (version: October, 2008)

[Hob20] J.D. Hobby, *METAPOST: A users's manual*, https://www.tug.org/docs/metapost/mpman.pdf (2020). (version: 2020)

[Knu86] D. Knuth, *METAFONT: The Program* (Addison Wesley Pub. Co, Reading, Mass, 1986).

[Ber14] C. Bersch, *Pst-intersect: Intersecting arbitrary curves*, https://github.com/cbersch/pst-intersect (2014). (version: March 16, 2014)

[Thu17] T. Thurston, *Drawing with Metapost*, https://github.com/thruston/Drawing-with-Metapost (2017). (version: March 2017)

[Tan22] T. Tantau, *The TikZ and PGF Packages: Manual for Version 3.1.9a*, https://github.com/pgf-tikz/pgf (2022). (version: March 29, 2022)

Mikael P. Sundqvist
mickep@gmail.com