Dotted and dashed lines in METAFONT

Jeremy Gibbons
Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand.
jeremy@cs.auckland.ac.nz

Abstract

We show how to draw evenly dotted and dashed curved lines in METAFONT, using *recursive refinement* of paths. METAPOST provides extra primitives that can be used for this task, but the method presented here can be used for both METAFONT and METAPOST.

Introduction

Knuth's METAFONT has powerful facilities for manipulating and drawing curves or 'paths'. These facilities are generally sufficient for METAFONT's primary intended purpose, namely drawing letters. However, METAFONT is also very well suited to producing technical diagrams; for this secondary purpose, METAFONT lacks a valuable facility—that of drawing evenly dotted and dashed curves. In this paper we show how to remedy this shortcoming, using the facilities that METAFONT does have available.

John Hobby's METAPOST is an adaptation of METAFONT for producing PostScript output rather than bitmaps. METAPOST *was* primarily intended for producing technical diagrams (Don Hosek reports Hobby as saying, 'Well, you *could* use it for generating characters, but I wouldn't recommend it'). METAPOST therefore provides an ingenious scheme for drawing dotted and dashed lines: an arbitrary picture can be used to generate a dash pattern for drawing paths. This scheme is not very general—there are reasonable dashed-line-like applications for which it does not work—but METAPOST also provides lower-level primitives `arclength` and `arctime` that are quite general. These primitives make the approach presented in this paper largely redundant for METAPOST, but it remains necessary for the 'core METAFONT' language.

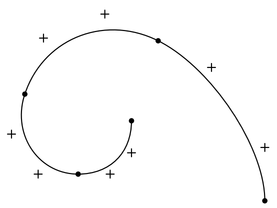
Throughout this paper, we use the term 'METAFONT' to refer to both Knuth's METAFONT and Hobby's METAPOST; we use the term 'METAPOST' to refer just to Hobby's METAPOST.

Cubic Bézier curves

METAFONT represents curved lines by *piecewise cubic Bézier curves*, which it calls 'paths'. These are discussed in Chapter 3 of the METAFONTbook; we summarize here all that is needed for the purposes of this paper. For most of this paper, we consider only *non-cyclic* paths; we discuss cyclic paths briefly at the end of the paper.

A path is specified by a sequence of *knots* and *control points*. The path runs from the first knot to the last knot, passing through each knot in turn. Between each pair of consecutive knots, there are two control points; the path leaves one knot in the direction of the next control point, and enters the next knot in the direction of the previous control point. A path with $n+1$ knots is said to have *length* n, and can be considered as a function from 'time' (i.e., the real numbers) between 0 and n inclusive to points in the plane; times that are natural numbers correspond to the knots, with time 0 the start of the path and time n the end. (Throughout this paper, we use the term 'length' as a measure of the number of knots in a path; it is always a natural number. In contrast, we use the term 'arc length' for the spatial distance covered in travelling along the path.)

For example, here is a spiral path of length 4, and the knots (dots) and control points (crosses) used to generate it.

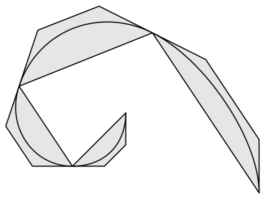


This path was drawn by the METAFONT code

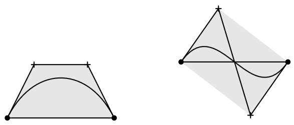
```
p := (90,0)  .. controls (90,20) and (70,50) ..
     (50,60) .. controls (30,70) and (7,61)  ..
     (0,40)  .. controls (-5,25) and (5,10)  ..
     (20,10) .. controls (32,10) and (40,18) ..
     (40,30);
draw p
```

METAFONT actually has sophisticated algorithms for choosing 'nice' control points given just the knots and possibly some other information, but that does not concern us here; whichever way the path is specified, METAFONT represents it internally as knots and control points.

An important property of piecewise cubic Bézier curves is that each *segment* of a path (i.e., a part between two consecutive knots) lies entirely within the 'convex hull' of —that is, the smallest convex polygon surrounding— the knots at either end and the control points in between. For example, here is the same spiral, with the convex hull surrounding each path segment shown shaded grey.



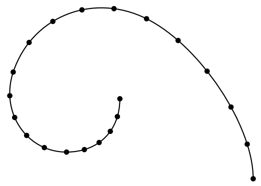
There are two general forms for a segment of a path: either the control points between the two knots are both on the same side of the 'chord' between the knots, or they are on different sides. These two cases are illustrated below.



Notice that, in either case, the arc length of the chord between the knots is no greater than that of the path, and the arc length of the 'control polygon' (consisting of three straight lines, from the first knot to the second via the control points) is no less than that of the path; this fact is important in what follows. Clearly, this property holds of paths as well as path segments. Also, the degenerate case in which both control points are on the same line as the knots yields a path that also lies entirely on that line; the arc lengths of the chord and control polygons again form lower and upper bounds on the arc length of the path.

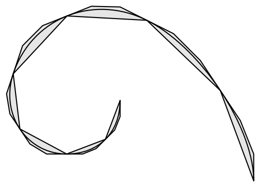
Evenly-spaced points on a path

The essential problem when it comes to drawing dotted or dashed lines is that points evenly separated in *time* along a path are not necessarily evenly separated in *space*. For example, here is the same spiral path as above, with 21 dots spaced evenly in time—that is, placed every $1/5$th of a time unit. Notice how the dots get closer as the curve gets tighter; a point moves at different 'speeds' in space as it progresses along the path evenly in time.



Producing evenly-spaced dots basically involves finding the arc length of a cubic Bézier curve. This is a difficult mathematical problem; it involves integrating the square root of a degree-four polynomial, which in turn can only be done analytically using 'elliptic integrals'—not one of the primitives provided by METAFONT.

Fortunately, there is a simple approximation method for finding the arc length; this method is the subject of this paper. The basis of the method is recursive *refinement* of the path, picking more and more knots on the path and hence using control points that are closer to the curve. For example, if we pick an extra knot half way (in time) between each pair of consecutive knots on the above spiral path, we get the following picture:



This spiral path is generated by the METAFONT code

```
path q;
q := subpath (0,0.5) of p
       for i := 1 step .5 until length p:
         & subpath(i-0.5,i) of p
       endfor;
```

The path itself has not changed (although it is now travelling at half of its original speed), but the chord and control polygons are much closer approximations to the curve. How good are these approximations? Gravesen (1993) shows that, under repeated recursive refinements, the average of the arc lengths of the chord and control polygons converges very quickly to the arc length of the path[1]. We use recursive refinement to get a sufficiently-close polygonal approximation to a path, and then divide that polygonal approximation up evenly in space. This yields (for example) the result below.

[1] In fact, the average of the arc lengths of the chord and control polygons under k recursive refinements converges to the arc length of the path as 16^{-k}; that is, the error decreases by a factor of 16 on every iteration. Gravesen also gives a general result for degree-n Bézier curves.

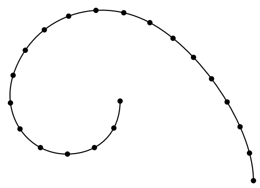
```
vardef chordpoly expr p =
  save i; numeric i;
  point 0 of p
  for i := 1 upto length p:
     -- point i of p
  endfor
enddef;

vardef controlpoly expr p =
  save i; numeric i;
  point 0 of p
  for i := 1 upto length p:
    -- postcontrol (i-1) of p
    -- precontrol i of p
    -- point i of p
  endfor
enddef;

vardef chordlen expr p =
  save i; numeric i;
  0
  for i := 1 upto length p:
    + abs(point i of p - point (i-1) of p)
  endfor
enddef;

vardef controllen expr p =
  chordlen (controlpoly p)
enddef;
```

Figure 1: The functions `chordlen` and `controllen`



Notice now that the dots are evenly spaced, even as the curve gets tighter.

Recursive refinement

In order to refine a path, we need first to be able compute lower and upper bounds to its arc length. The functions `chordlen` and `controllen`, which return the arc lengths of the chord and control polygons, are defined in Figure 1. (The METAFONT expression `point i of p` returns the position of the path at time i; for natural i, `postcontrol i of p` returns the first control point after knot i, and similarly, `precontrol i of p` returns the last control point before knot i.)

Given the lower bound `chordlen p` and upper bound `controllen p` to the arc length of a path p, recursive refinement is straightforward. If the bounds are sufficiently close, we return just p; otherwise we split p into two halves (time-wise), independently refine the two halves, and join the results back together. We use a multiplicative rather than additive test for 'sufficiently close', so that if a number of subpaths are independently 'sufficiently refined' then their concatenation will also be. Note that splitting the path in two doesn't double the length, as we did in our earlier spiral example; it only adds zero or one more knot (depending on whether or not `length p` is even). However, it does give the advantage of *adaptive* refinement—nearly straight parts of the path are not refined as much as very wiggly parts.

```
numeric tol; tol := eps;
vardef refine expr p =
  if (controllen p)<=(1+tol)*(chordlen p):
    p
  else:
    (refine (subpath(0, length p/2) of p)) &
    (refine (subpath(length p/2,length p) of p))
  fi
enddef;
```

Marking a path evenly

Having refined the path p, we still need to divide it into equal-sized chunks; that is, we need to find a sequence of times $t_0, \ldots, t_r$ such that the arc length between points t_i and t_{i+1} of p for each i is some fixed given distance d. However, we now have a polygonal path `chordpoly(refine p)` which very closely approximates p. It is straightforward (if a little messy) to find the times that divide this polygonal approximation into chunks of arc length d; we simply use those same times for the curved path p.

The code for the function `markedevery` is given in Figure 2. The function takes in a path p and a distance d, and returns the sequence of times that divide the chord polygon of p evenly into chunks of arc length d. This function should be called only on a path which is 'very nearly' polygonal—one that is actually polygonal, or perhaps the result of refining another path.

The program maintains variables t, the 'current time', which increases from 0 to the length (in time) of p and is always at an integer value (in fact, equal to `knot`) at the start of the outer loop body, and d_{next}, which is the distance from the 'current point' (point t of p) until the next mark. Each path segment is considered in turn, the time counter t advancing as required along it. Note that the first and last 'chunks' taken from a segment may be shorter than d.

```
secondarydef p markedevery d =
  begingroup
    save q; path q; q := (0,0); % for result
    save dnext; numeric dnext; dnext := d;
    save seglength; numeric seglength;
    save knot; numeric knot;
    save t; numeric t; t:=0;
    save dt; numeric dt;
    for knot := 0 upto length p - 1:
      seglength := abs (point (knot+1) of p
                        - point knot of p);
        % arc length of this segment
      if seglength > 0:
        forever:
          dt := dnext / seglength;
            % time to next mark (if this segment)
          exitif t+dt > knot+1;
            % exit if next mark not on this seg
          t := t+dt; % move forwards...
          q := q -- (t,0); % ... & put mark here
          dnext := d; % next mark is d away
        endfor
        % now t <= knot+1 < t+dt
        dnext := dnext - (knot+1-t)*seglength;
          % put leftover towards next mark
        t := knot+1;
      else: % empty seg (coincident knots)
        t := t+1;
      fi
    endfor;
    q % return time sequence we've built up
  endgroup
enddef;
```

Figure 2: The function `markedevery`

```
def drawdotted (expr p, d) =
  save refined, marks, i;
  path refined, marks; numeric i;
  refined := refine p;
  marks := refined markedevery d;
  for i := 0 upto length marks:
    drawdot
      point (xpart(point i of marks)) of refined;
  endfor
enddef;
```

Figure 3: The procedure `drawdotted`

METAFONT has no 'list' type that can be used to return the sequence of times $t_0, \ldots, t_r$, so we return the path $(t_0, 0) -- \cdots -- (t_r, 0)$ instead; this path is built up in the variable `q`.

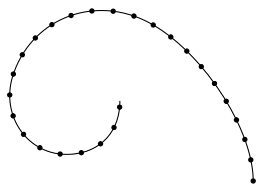
Dotted and dashed lines

The function `markedevery` does the work of marking a path evenly in space; as described above, to draw a dotted path we first refine it, and mark the refined polygonal approximation evenly instead. The procedure `drawdotted` is defined in Figure 3.

For example, we can place dots every eight units in space on our spiral path as follows:

```
pickup pencircle scaled 0.5;
draw p;
pickup pencircle scaled 2;
drawdotted (p, 8);
```

This yields the picture



Notice that there is no guarantee that the last dot will be at the end of the path. To ensure that this is the case, we must choose the dot spacing to divide evenly into the arc length of the path. Fortunately, the average of the arc lengths of the chord and control polygons makes a very good estimate of the arc length, as discussed above. The evenly-dotted spiral on page 3 with the 21st dot exactly at the end of the path was drawn with the commands

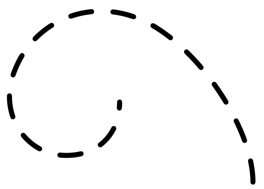
```
pickup pencircle scaled 0.5;
draw p;
pickup pencircle scaled 2;
path q; q := refine p;
drawdotted (p,
  .5[chordlen q,controllen q]/(20*(1+eps)));
```

(Notice that we have to scale down the 'ideal' dot spacing by a factor of `1+eps`, to ensure that the last dot is just on rather than just off the end of the path in case of rounding errors.)

Dashed lines can be drawn using pretty much the same approach. Here is a simple-minded macro to do it.

```
def drawdashed (expr p, d) =
  save refined, marks, i;
  path refined, marks; numeric i;
  refined := refine p;
  marks := refined markedevery d;
  if (length marks) mod 2 = 0:
    marks := marks -- (length refined,0);
  fi
  for i := 0 step 2 until length marks - 1:
    draw subpath(xpart (point i of marks),
      xpart (point (i+1) of marks)) of refined;
  endfor
enddef;
```

For example, with the same spiral path and $d = 8$ we get



Notice that we add an extra mark at the end of the path (yielding a partial final dash) if the number of marks is odd.

A more elaborate approach would allow differing dash and gap sizes, and displacing the dashes. This could be done by using the greatest common divisor of the various distances to mark the path, or perhaps by altering the `markedevery` function to take several distances as arguments.

Cyclic paths

The same recursive refinement technique works just as well for cyclic paths; in fact, the only change that is needed is to the function `refine`. When we split a path into two halves, we need to remember whether the path is cyclic, recombining the halves as a cycle if so. The code for this version of `refine` is given in Figure 4.

Iterative non-adaptive refinement

The recursive refinement technique described here is quite elegant, but it can cause METAFONT to run out of stack space on very wiggly paths. An iterative approach can avoid this, at the cost of not easily providing adaptive refinement; we simply repeatedly double the length until the lower and upper bounds

```
vardef refine expr p =
  if (controllength p)<=(1+tol)*(chordlength p):
    p
  else:
    (refine (subpath(0, length p/2) of p)) &
    (refine (subpath(length p/2,length p) of p))
      if cycle p: & cycle fi
  fi
enddef;
```

Figure 4: `refine` for cyclic paths

```
vardef refine expr p =
  save q; path q; q := p;
  forever:
    exitif (controllen q)<=(1+tol)*(chordlen q);
    q := subpath (0,0.5) of q
           for i := 1 step .5 until length q:
             & subpath(i-0.5,i) of q
           endfor
           if cycle q: & cycle fi;
  endfor;
  q
enddef;
```

Figure 5: An interative version of `refine`

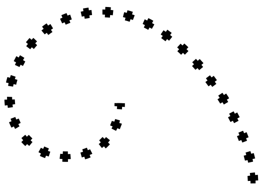
on the arc length are sufficiently close. The code is given in Figure 5.

An unusual application

As we mentioned in the introduction, Hobby's METAPOST does provide primitives for drawing dotted and dashed lines. You can do some surprising things with dashes; for example, you can draw crosses along a path by drawing the path twice, once with long thin dashes and once with short fat dashes.

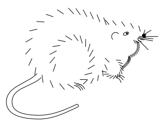
```
interim linecap:=butt;
draw p dashed dashpattern(on 6 off 6)
   withpen pencircle scaled 2;
draw p dashed dashpattern(off 2 on 2 off 8)
   withpen pencircle scaled 6;
```

(The assignment to `linecap` produces square, rather than rounded, ends to lines.) Unfortunately, although METAPOST generates good PostScript from such constructions, bugs in many PostScript interpreters make them come out wrong. For example, the following picture should consist of crosses, but on some PostScript interpreters some of the crosses turn out mushroom-shaped.



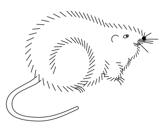
Still, there are some things that cannot be done with METAPOST's dash primitives, but can be done with the techniques described here. We conclude this paper with one such application. (In fact, this application was the original motivation for the author's interest in the topic.) Note that this problem can also be solved using METAPOST's `arclength` and `arctime` primitives, but then the solution is not portable to 'core METAFONT'.

Several years ago, while a PhD student, the author used to play in a jazz band called the *Mississippi Muskrats*, and he endeavoured to produce a logo for posters for the band. This logo included a picture of a muskrat, for which a 'furry' effect was obtained by drawing diagonal lines across a path. The first attempt at drawing a furry muskrat used the same time interval for every path, and yielded a bushy throat and a threadbare back:

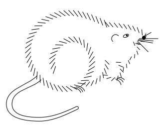


(The author makes no claims for the artistic merit of these drawings.)

The second attempt used different time intervals for different paths, and was much better; still however, the muskrat's back suddenly gets hairier about halfway from the tail to the neck, and of course there's the hassle of choosing all those different numbers.



The third attempt used the method described in this paper, and gave a much healthier-looking muskrat:



Acknowledgements

The author would like to thank Jens Gravesen for his very elegant paper (Gravesen 1993), and Alan Hoenig, Geoffrey Tobin and several other people who have put up with discussions of this topic electronically and in person over the last few years.

References

Gravesen, Jens. "Adaptive Subdivision and the Length of Bézier Curves". Technical Report 472, The Danish Center for Applied Mathematics and Mechanics, Technical University of Denmark, 1993.

◇ Jeremy Gibbons
Department of Computer Science,
University of Auckland,
Private Bag 92019, Auckland,
New Zealand.
jeremy@cs.auckland.ac.nz