Computing the area and winding number for a Bézier curve

Bogusław Jackowski

1 Introduction

Why would we want to compute the area or winding number for a given (closed) Bézier curve? The general answer is: in computational graphics, various measures of graphical objects may prove useful.

For example, MetaPost was equipped with a very important function, missing from Metafont: `arclength`, which computes the length of a given arc. A typical problem that can be easily solved using this function is placing uniformly spaced text along a curve.

In my font application, I needed a function that calculates a distance between two curves — this feature could be used to compute an approximation of a multi-node Bézier curve by a single Bézier arc (i.e., for simplifying curves). Another operation I needed is a Boolean function telling whether a given curve is embedded in another curve.

These operations are missing from both Metafont and MetaPost, although they are feasible in Metafont due to its bitmap operations.

A useful distance between curves `a` and `b` can be computed as the number of pixels that receive a nonzero value when filling the curve `a--b--cycle` (for a given resolution). Also, checking two closed curves `a` and `b` for mutual embedding could be calculated (again, for a given resolution) by filling the curve `a` and unfilling the curve `b` — if no pixels with a positive value remain, it means that `a` is embedded in `b`.

It should be emphasized that the result depends on the resolution, which can certainly be considered a drawback.

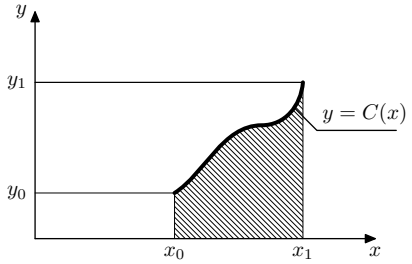
In MetaPost, bitmap operations are unavailable, hence computing the distance between curves and checking the mutual embedding of curves cannot be done simply. However, a distance between two non-intersecting curves, `a` and `b`, can be computed as the absolute value of the area surrounded by `a--b--cycle`; if the curves intersect, one has to find all intersections and compute the area for all subcycles, which is a fairly complex task (due to numerical instability).

On the other hand, a winding number — for non-touching curves — can be used for determining the mutual position of the two curves (as explained in the section below).

While bitmap operations are practically insensitive to the tangency of curves, the algorithm for computing the winding number presented here crucially is. Nevertheless, these MetaPost algorithms for computing the area and winding number may be considered counterparts of the relevant bitmap-based Metafont algorithms. Of course, I would still gladly welcome building bitmap operations into MetaPost.

2 Area enclosed by a cyclic Bézier spline

The area between the graph of a function $x \mapsto (x, C(x))$ and the x-axis is shown as the hatched region in this figure:



It can be computed as the integral

$$\int_{x_0}^{x_1} C(x)dx \tag{1}$$

If the curve is given parametrically, i.e., $t \mapsto (C^x(t), C^y(t))$, the integral (1) can be rewritten (by substituting $x = C^x(t)$, $x_0 = C^x(t_0)$, $x_1 = C^x(t_1)$, $C(x) = C(C^x(t)) = C^y(t)$, and $dx = \frac{dC^x(t)}{dt}dt$) as

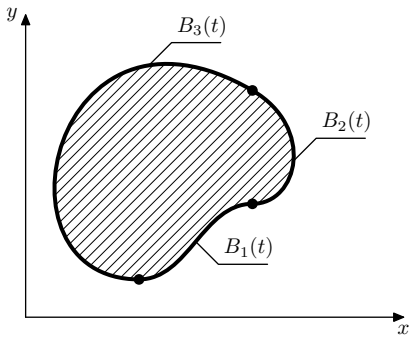
$$\int_{t_0}^{t_1} C^y(t)\,\frac{dC^x(t)}{dt}dt \tag{2}$$

Furthermore, if $t_0 \neq t_1$ and

$$(C_x(t_0), C_y(t_0)) = (C_x(t_1), C_y(t_1)),$$

i.e., the curve is cyclic, the integral (2) yields the area surrounded by the curve.

Assume that the cyclic curve is a spline composed of Bézier arcs B_1, $B_2, \ldots, B_n$ (each defined for $0 \leq t \leq 1$). The area of the region surrounded by the spline shown here:



is the sum of integrals:

$$\sum_{i=1}^{n} \int_0^1 B_i^y(t)\frac{dB_i^x(t)}{dt}\,dt$$

In the following, I'll skip the index i, as calculations are exactly the same for each i; the functions $B(t) = (B^x(t), B^y(t))$ are third-degree polynomials:

$$B(t) = b_0(1-t)^3 + 3b_1(1-t)^2\,t + 3b_2(1-t)\,t^2 + b_3t^3$$

where $b_0 = (b_0^x, b_0^y)$, $b_1 = (b_1^x, b_1^y)$, $b_2 = (b_2^x, b_2^y)$, $b_3 = (b_3^x, b_3^y)$ are points in the plane; b_0, b_3 are the nodes and b_1, b_2 are the control points of the Bézier arc B.

The computation of the antiderivative of the function $B^y(t)\frac{dB^x(t)}{dt}$ (a fifth-degree polynomial) is an elementary task (actually, it suffices to know that a derivative of t^n is nt^{n-1} and, thus, the integral of t^n is $\frac{1}{n+1}t^{n+1}$). Skipping tedious calculations, I'll present the final formula:

$$\begin{aligned}20\int_0^1 B^y(t)\frac{dB^x(t)}{dt}\,dt = {}& (b_1^x-b_0^x)(10b_0^y+6b_1^y+3b_2^y+b_3^y)\\ &+(b_2^x-b_1^x)(4b_0^y+6b_1^y+6b_2^y+4b_3^y)\\ &+(b_3^x-b_2^x)(b_0^y+3b_1^y+6b_2^y+10b_3^y)\end{aligned}$$

This formula stemmed from the discussion between Daniel Luecking and Laurent Siebenmann on the Metafont/MetaPost discussion list (`metafont@ens.fr`, 2000; presently the MetaPost discussion list is hosted by TUG — `http://lists.tug.org/metapost`). Luecking made a crucial observation that three real multiplications per Bézier arc sufficient to compute the area surrounded by a Bézier spline; division of the whole sum by 20 is a constant cost and thus can be neglected. Integer multiplication can be replaced by operations usually faster than real multiplication (e.g., $10a = 8a + 2a$, $8a = a$ shifted left by 3 bits, $2a = a$ shifted left by 1 bit).

Of course, such an optimization of the arithmetic operations makes sense only in a "production" implementation of the algorithm. The implementation at the level of Metafont/MetaPost macros can be neither efficient nor precise. Nevertheless, the following code may sometimes prove useful:

```
% p is a B\'ezier segment; result = \int y dx
vardef area(expr p) =
  save xa, xb, xc, xd, ya, yb, yc, yd;
  (xa,20ya) = point 0 of p;
  (xb,20yb) = postcontrol 0 of p;
  (xc,20yc) = precontrol 1 of p;
  (xd,20yd) = point 1 of p;
    (xb-xa)*(10ya + 6yb + 3yc +   yd)
   +(xc-xb)*( 4ya + 6yb + 6yc +  4yd)
   +(xd-xc)*(  ya + 3yb + 6yc + 10yd)
 enddef;

% P is a cyclic path; result = area of interior
vardef Area(expr P) =
 area(subpath (0,1) of P)
  for t=1 upto length(P)-1:
    + area(subpath (t,t+1) of P) endfor
enddef;
```

Observe that the macro *Area* computes a signed area: negative for counterclockwise-oriented curves, and positive for clockwise-oriented ones. As a consequence, a non-trivial curve with self-intersection(s) (e.g., eight-shaped) may surround a region with the area equal to zero.

Observe also that the calculations can be carried out with respect to the y-axis, thus the following code

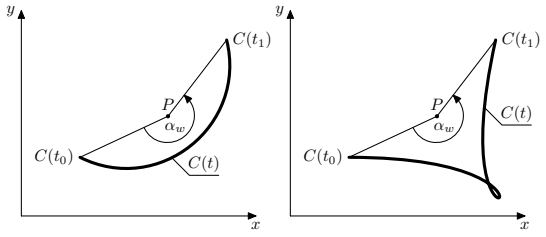
```
% p is a B\'ezier segment; result = \int y dx
vardef area(expr p) =
 save xa, xb, xc, xd, ya, yb, yc, yd;
 (-20xa,ya) = point 0 of p;
 (-20xb,yb) = postcontrol 0 of p;
 (-20xc,yc) = precontrol 1 of p;
 (-20xd,yd) = point 1 of p;
   (yb-ya)*(10xa + 6xb + 3xc +   xd)
  +(yc-yb)*( 4xa + 6xb + 6xc +  4xd)
  +(yd-yc)*(  xa + 3xb + 6xc + 10xd)
enddef;

% P is a cyclic path; result = area of interior
vardef Area(expr P) =
 area(subpath (0,1) of P)
  for t=1 upto length(P)-1:
    + area(subpath (t,t+1) of P) endfor
enddef;
```

will yield the same results as the former (within the accuracy of rounding errors).

3 A winding number for Bézier splines

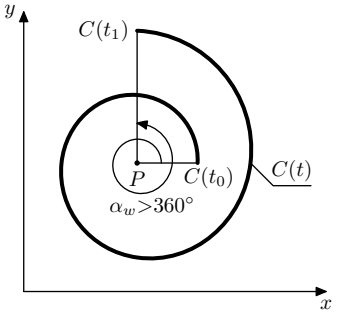
Assume that we are given a point P in the plane and the planar curve $C(t)$ defined for $t_0 \le t \le t_1$. The total angle encircled by the radius $P\,C(t)$ as t runs from t_0 to t_1 we will call the *winding angle* and denote by α_w:



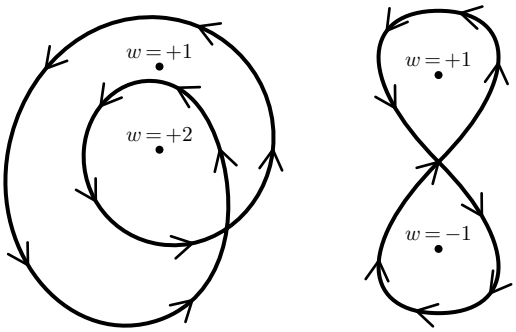
This winding angle is insensitive to certain local properties of the curve $C(t)$ (e.g., local loops): in the figures above, the *winding* angle is the same in both cases (assuming the same points P, $C(t_0)$ and $C(t_1)$).

The winding angle is positive if the point P lies to the right with respect to the point traversing the curve, and negative otherwise.

Of course, the absolute value of a winding angle can be larger than 360°:



For cyclic curves, the *winding angle* is always a multiple of 360°, i.e., $\alpha_w = 360° \, w$, where w is an integer. This entity w is called the *winding number* (for a given point and curve).



In the following, we will focus our attention on cyclic Bézier splines.

The idea of the algorithm computing the *winding number* for Bézier splines is due to Laurent Siebenmann (`metafont@ens.fr`, 2000; now at `http://lists.tug.org/metapost`). Siebenmann's solution, however, was MetaPost-oriented — it exploited heavily the operation *arctime*, available in MetaPost but unavailable in Metafont. Below, I'll present an algorithm based on the same idea but referring to more elementary properties of a Bézier segment.

For a given point P and a Bézier spline C, we will try to find the *winding angle* by measuring the *winding angles* for a discrete series of time points. First, we will try to measure the angles between nodes 0, 1, 2, ..., n of the spline C. If the relevant Bézier segments are appropriately short, the sum of the angles yields the total *winding angle*. The problem arises when the Bézier arc is too long — see, e.g., the leftmost panel of the first figure (the angle $C(t_0)\,P\,C(t_1)$ equals $360° - \alpha_w$).

The main observation of Siebenmann is as follows: if the length of the subarc $C(t)$ for $t_0 \le t \le t_1$ is shorter than the length of the longer of the radii $P\,C(t_0)$ and $P\,C(t_1)$, than we can safely assume that the (acute) angle between $P\,C(t_0)$ and $P\,C(t_1)$ is the *winding angle*. In fact, we do not need to know the exact length of the arc — an approximation suffices. If B_a, B_b, B_c, and B_d are the points defining a Bézier arc B (i.e., B_a and B_d are its nodes, B_b and B_c are its control points), then

$$|B_a\,B_b| + |B_b\,B_c| + |B_c\,B_d| \ge |B|$$

(with $|\ldots|$ denoting the length of an interval and the length of a Bézier arc). In other words, we can safely use the left-hand side of the above inequality instead of the true value of the arc length in the computation of the winding angle or winding number.

The algorithm can be expressed in pseudo-code as follows:

input: a point P and a Bézier spline B,
  consisting of segments B_1, $B_2, \ldots, B_n$
output: α_w — the winding angle for P and B
procedure *windingangle*(P, B)
  if B is a single segment
    let B_a, B_b, B_c, B_d be the consecutive
      control nodes of the segment B
    if $\min(|P\,B_a|, |P\,B_d|) <$ *assumed minimal dist.*
      exit (P almost coincides with B,
        winding angle incalculable)
    fi
    if $|B_a\,B_b| + |B_b\,B_c| + |B_c\,B_d|$
      $> \max(|P\,B_a|, |P\,B_d|)$
      return *windingangle*$(P, B(0, 1//2))$
        $+$*windingangle*$(B(1//2, 1))$
    else
      return angle α between the radii $P\,B_a$
        and $P\,B_d$ $(-90° < \alpha < 90°)$
    fi
  else
    return *windingangle*$(P, B_1) + \ldots$
      $+$*windingangle*(P, B_n)
  fi
end

A possible MetaPost/Metafont implementation:

```
% B is a B\'ezier segment
vardef mock_arclength(expr B) =
 % |mock_arclength(B)>=arclength(B)|
 length((postcontrol 0 of B)-(point 0 of B))
 + length((precontrol 1 of B)-(postcontrol 0 of B))
 + length((point 1 of B)-(precontrol 1 of B))
enddef;

% P is a point, B is a B\'ezier spline
vardef windingangle(expr P,B) =
 if length(B)=1: % single segment
  save r,v;
  r0=length(P-point 0 of B);
  r1=length(P-point 1 of B);
  if (r0<2eps) and (r1<2eps):
   % MF and MP are rather inaccurate, return 0
   errhelp "Not advisable to continue.";
   errmessage "windingangle: point almost "
```

```
    & "coincides with B\'ezier segment (dist="
    & decimal(min(r0,r1)) & ")";
  0
 else:
  % v denotes both length and angle
  v := mock_arclength(B);
  % possibly too long B\'ezier arc?
  if (v>r0) and (v>r1):
   windingangle(P, subpath (0, 1/2) of B)
   + windingangle(P, subpath (1/2, 1) of B)
  else:
   v := angle((point 1 of B)-P)  %difference
        - angle((point 0 of B)-P);
   if v >= 180: v := v-360; fi   %normalize
   if v < -180: v := v+360; fi
   v                             %return
  fi
 fi
else: % multisegment spline
 windingangle(P,subpath (0,1) of B)
  for i:=1 upto length(B)-1:
    + windingangle(P,subpath (i,i+1) of B)
  endfor
fi
enddef;
```

Although the returned angle (line marked '`%return`' above) is acute, the difference of the component angles (lines at '`%difference`') can be outside the interval $\langle -180°, 180°\rangle$; hence the normalization (lines at `%normalize`).

If the operation *windingnumber* is needed for some reason, it can be implemented trivially:

```
% P is a point, B is a B\'ezier spline
vardef windingnumber (expr P,B) =
  windingangle(P,B)/360
enddef;
```

The operations *windingangle* or, equivalently, *windingnumber* can be used, e.g., for determining the mutual position of two non-intersecting cyclic curves (whether one is embedded inside the other or not):

```
tertiarydef a inside b =
 if path a:
 % |and path b|; |a| and |b| must be cyclic and
 % must not touch each other
  begingroup
   save a_,b_;
   (a_,b_) = (windingnumber(point 0 of a,b),
              windingnumber(point 0 of b,a));
   (abs(a_ - 1) < eps) and (abs(b_) < eps)
  endgroup
 else: % |numeric a and pair b|
  begingroup
   (a>=xpart b) and (a<=ypart b)
  endgroup
 fi
enddef;
```

Postscriptum

In some cases, another definition, equivalent to the one formulated above, may be useful (this formulation, given below without a proof of equivalence, is a slightly edited excerpt from Siebenmann's message):

Assume that we are given a curve C and point P. Choose at random a line segment emanating from the point P to the point W, with W outside the bounding box of C and P. Inductively examine the intersection points Q of PQ with C. Supposing these points Q are all "nondegenerate" intersections, they are also finite in number, and a sign $+1$ or -1 is associated with each. Nondegenerate means that Q is a smooth point of c and the tangent vector T to C at Q is not parallel to PQ, and that Q is not a point where C crosses itself. The sign to use is the sign of the wedge product '$(Q-P)$ wedge T', i.e.,

$$(Q-P)\cdot(T \text{ rotated } -90)$$

The sum of the signs is the *winding number*.

It is a probabilistic theorem that degenerate intersections will rarely be met.

◇ Bogusław Jackowski
Gdańsk, Poland
b_jackowski (at) gust dot org
dot pl