Computers & Graphics 38 (2014) 365–373

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Special Section on CAD/Graphics 2013

A continuation algorithm for planar implicit curves with singularities

Abel J.P. Gomes *

Instituto de Telecomunicações, Universidade da Beira Interior, Portugal

ARTICLE INFO

Article history:
Received 4 August 2013
Received in revised form
6 November 2013
Accepted 6 November 2013
Available online 21 November 2013

Keywords:
Implicit curves
Continuation methods
Newton corrector
Singularities
Critical points

ABSTRACT

Continuation algorithms usually behave badly near to critical points of implicitly defined curves in $\mathbb{R}^2$, i.e., points at which at least one of the partial derivatives vanishes. Critical points include turning points, self-intersections, and isolated points. Another problem with this family of algorithms is their inability to render curves with multiple components because that requires finding first a seed point on each of them. This paper details an algorithm that resolves these two major problems in an elegant manner. In fact, it allows us not only to march along a curve even in the presence of critical points, but also to detect and render curves with multiple components using the theory of critical points.

1. Introduction

A classical problem found in computer graphics is to translate an implicit expression (e.g., the one representing the hyperbola $9x^2-4y^2-72x+8y+176=0$) into the graphical form. By algebraic manipulation of this kind of expressions with two variables, it is possible to solve for y in terms of x, from where we are able to plot points (x,y) on the curve. However, this approach does not generalize easily to high order polynomials and is impracticable for general analytic expressions such as, for example, transcendentals of the form $y \sin x + x \cos y = 1$. Curves defined by general analytic expressions are called implicitly defined curves or, simply, implicit curves.

More formally, a planar implicit curve $\mathcal{C}$ is the zero set of some real function f from $\mathbb{R}^2$ to $\mathbb{R}$, that is, $\mathcal{C}=\{\mathbf{x}\in\mathbb{R}^2 : f(\mathbf{x})=0\}$. The main categories of algorithms for rendering implicit curves are the following:

- *Implicit-to-parametric conversion*: The generative nature of parametric curves makes them suited to rendering; hence, the implicit-to-parametric conversion [1,2]. However, this technique is not always feasible because many implicit curves cannot be globally parameterized, regardless of whether they are smooth or not.
- *Curve continuation*: The idea behind curve continuation (or tracking) algorithms is to generate a sequence of points on the curve in a way similar to parametric curves (see, for example, [3–5]). In computer graphics, Bresenham's algorithm for rendering circles was the former algorithm to adopt this approach. These methods are attractive because they confine the calculation of the next point to the neighborhood of current point. But some components of the curve may be missed out unless we are capable of finding a starting point on each component. Even worse is the fact that tracking algorithms may break down on singularities.
- *Space subdivision*: The leading idea of this approach is to subdivide the rectangular domain $D\in\mathbb{R}^2$ into subregions in the attempt to find the curve, discarding then those regions that are not crossed by the curve. The recursive subdivision terminates as long as the set of small regions that approximate the curve satisfy a specific criterion (curvature, distance, etc.) [2,6–8]. Subdivision algorithms are often combined with interval arithmetic [9,10], affine arithmetic [11,12], or variants [13] in order to guarantee that the traced curve exhibits its topology correctly.
- *Symbolic computation*: Unlike continuation and subdivision algorithms that are based on floating-point computations (i.e., numerical algorithms) to sample curves, symbolic algorithms are exact because they handle symbols and symbolic expressions. In the literature, we find a number of these algorithms for algebraic curves, but only a few apply to trigonometric curves [14]. As expected, we also find symbolic algorithms to determine the topological type of a curve [15–17], as well as singularities [18,19], and roots [20–24]. But, symbolic algorithms are slower than numerical algorithms.

The algorithm described in this paper fits in the category of *continuation* algorithms. Thus, the algorithm produces a piecewise linear approximation (e.g., a number of polylinearized contours or components) of an implicit curve on the fly and in a progressive manner.

* Tel.: +351 275 242081.
E-mail address: agomes@di.ubi.pt

0097-8493/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cag.2013.11.006

The algorithm applies to analytic curves, including algebraic curves and transcendental curves (e.g. transcendental and exponential functions). Its main *contributions* lie in the fact that the algorithm is capable of dealing with singular points, detecting and tracing multiple components of a given curve, regardless of whether they are sign-variant components or not, as well to prevent drifting and looping phenomena in tracing implicit curves.

The remainder of this paper is organized as follows. Section 2 approaches implicit curves as level sets of surfaces in $\mathbb{R}^3$ in order to provide insight on the geometry of curves, providing then details about the mathematics and the core of the algorithm. Section 3 extends the algorithm to non-manifold implicit curves, including those having 0- and 1-dimensional singular sets and multiple components. Section 4 provides some relevant details concerning computing assets and time performance. Finally, some conclusions are drawn in Section 5.

2. The algorithm for manifold curves

An implicit curve $f(x,y)=c$ in $\mathbb{R}^2$ can be thought of as a contour resulting from the intersection of a plane $z=c$ and a surface $z=f(x,y)$ in $\mathbb{R}^3$, where $f:\mathbb{R}^2\to\mathbb{R}$ is a real function of two real variables. Taking into consideration that every value of the function $z=f(x,y)$ can be represented as a point height value $z=f(x,y)$ above of the point (x,y) in the plane XY, we easily see that the result is a surface with ups (hills) and downs (valleys).

The initial step in tracing an implicit curve $f(x,y)=c$ is to find a seed point of the level curve with height c, after which we determine a sequence of curve points that allow us to polylinearize and draw the corresponding contour. The computation of every new curve point involves two steps: *predictor* and *corrector*. That is, the current point is used to predict a point near to the curve, being the predicted point then corrected to a new point of the curve, which is the next point in the process of tracking the curve. The prediction step uses the tangent vector obtained from the gradient vector, whereas the correction step makes usage of a numerical zero finder (e.g., Newton corrector).

Let us then start the tracking of the curve, taking into account that $d_x(x,y)$ and $d_y(x,y)$ are the discrete partial derivatives given by forward differences. Moving a x-step Δ_x from (x,y) to $(x+\Delta_x,y)$ results in a change in the z-direction given by $\Delta_x\, d_x(x,y)$; analogously, a y-step from $(x+\Delta_x,y)$ to $(x+\Delta_x,y+\Delta_y)$ originates a change in the z-direction given by $\Delta_y d_y(x+\Delta_x,y)$. Now, summing up the above two height changes from (x,y) to $(x+\Delta_x,y+\Delta_y)$ yields a point either under or above the plane $z=c$ where our curve lies in. Taking this point back to this level plane in z-direction results in a null overall change in z-direction, so we have

$$\Delta_x d_x(x,y)+\Delta_y d_y(x+\Delta_x,y)=0. \tag{1}$$

But, using Eq. (1), we cannot express Δ_x in terms of Δ_y. To solve this problem, we have to assume that $d_y(x+\Delta_x,y)$ and $d_y(x,y)$ are pretty close so that we are allowed to replace $d_y(x+\Delta_x,y)$ by $d_y(x,y)$ in (1) to get an efficient, but possibly worse, approximation as follows:

$$\Delta_x\, d_x(x,y)+\Delta_y\, d_y(x,y)=0. \tag{2}$$

from which we have

$$\Delta_x=-\frac{d_y(x,y)}{d_x(x,y)}\Delta_y \quad\text{and}\quad \Delta_y=-\frac{d_x(x,y)}{d_y(x,y)}\Delta_x. \tag{3}$$

In order to have a neat control on the tracking procedure, we have to choose either x-direction or y-direction that minimises the distance to the curve. We know this after computing the absolute values of the components $d_x(x,y)$, $d_y(x,y)$ of the discrete gradient. The correct axial direction is the one given by $\min(|d_x(x,y)|,|d_y(x,y)|)$; it only remains to know whether it is forward or backward. Looking at (3), and letting

$$A=-d_y(x,y) \quad\text{and}\quad B=d_x(x,y) \tag{4}$$

we end up having

$$\Delta_x=\frac{A}{B}\Delta_y \quad\text{and}\quad \Delta_y=\frac{B}{A}\Delta_x. \tag{5}$$

Therefore, if $|A|>|B|$, then we walk along the x-direction with a pre-defined step Δ. To make sure we walk in the same direction as A, we put

$$\Delta_x=\Delta\ \mathrm{sign}(A) \quad\text{and}\quad \Delta_y=\frac{B}{A}\Delta_x. \tag{6}$$

Otherwise, and assuming that $B\neq 0$, we take

$$\Delta_y=\Delta\ \mathrm{sign}(B) \quad\text{and}\quad \Delta_x=\frac{A}{B}\Delta_y. \tag{7}$$

The new predicted point $(x+\Delta_x,y+\Delta_y)$ of the curve takes over the current point (x,y). Such a predicted point is then corrected using the discrete Newton corrector.

Unlike other predictor–corrector algorithms, the computation of the discrete tangent vector (A,B) (steps 4–5 of Algorithm 1) is not used to determine the next predicted point (i.e., the endpoint of the tangent vector) *directly*, but just to determine how to walk in both x- and y-directions, i.e., to determine the next stair step that better approximates the curve. Therefore, during the predictor stage, our algorithm behaves as a Bresenham-like algorithm in the sense that, depending on the slope of the tangent at a curve point, it steps forward first in the x-direction and then in the y-direction, or vice-versa.

The corrector stage makes usage of a discrete Newton corrector (i.e., which uses discrete derivatives) to ensure that the algorithm does not break down at critical points. This is illustrated in Algorithm 2, where the Newton correction that takes a point toward a curve point is carried out in either the x-direction or the y-direction (cf. lines 6 and 11 in Algorithm 2), resulting in a staircase-like walking along the curve; hence the name of Algorithm 1. The input variables f, D, and Δ of this algorithm stand for the curve function, the boxed domain, and the step length, respectively. Interestingly, Algorithm 1 is capable of rendering some curves with singularities as, for example, the cuspidal curve shown in Fig. 3(a), but not non-manifold curves featuring nodes or self-intersections as those shown in Fig. 3(b) and (c).

Algorithm 1. PC algorithm for manifold curves.

```
1:  Procedure Staircase (f,D, Δ)
2:    Determine one point (x,y) on the curve
3:    while (x,y) ∈ D do
4:      A ← −d_y(x,y)        ▷ first tangent component
5:      B ← d_x(x,y)         ▷ second tangent component
6:      PredictorCorrector(x,y,f,A,B, Δ)
7:      LineTo(x,y)          ▷ draws a line segment
8:    end while
9:  end procedure
```

Algorithm 2. Predictor–corrector.

```
1:  procedure PredictorCorrector(x,yf,A,B, Δ)
2:    if |B| < |A| then
3:      Δ_x ← Δ . sign(A)
4:      x ← x+Δ_x            ▷ predicted point (x,y)
5:      y ← y+Δ_x·B/A        ▷ see (6)
6:      y ← NewtonY(x,y,f)   ▷ corrector
7:    else
8:      Δ_y ← Δ.sign(B)
9:      y ← y+Δ_y            ▷ predicted point (x,y)
```