Efficient Rasterization of Implicit Functions

Torsten Möller and Roni Yagel

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio

{moeller, yagel}@cis.ohio-state.edu

Abstract

Implicit curves are widely used in computer graphics because of their powerful features for modeling and their ability for general function description. The most popular rasterization techniques for implicit curves are space subdivision and curve tracking. In this paper we are introducing an efficient curve tracking algorithm that is also more robust then existing methods. We employ the Predictor-Corrector Method on the implicit function to get a very accurate curve approximation in a short time. Speedup is achieved by adapting the step size to the curvature. In addition, we provide mechanisms to detect and properly handle bifurcation points, where the curve intersects itself. Finally, the algorithm allows the user to trade-off accuracy for speed and vice a versa. We conclude by providing examples that demonstrate the capabilities of our algorithm.

1. Introduction

Implicit surfaces are surfaces that are defined by implicit functions. An implicit function f is a mapping from an n dimensional space R^n to the space R, described by an expression in which all variables appear on one side of the equation. This is a very general way of describing a mapping. An n dimensional implicit surface S is defined by all the points p in R^n such that the implicit mapping f projects p to 0. More formally, one can describe S by

$$S_f = \{p \mid p \in R^n, f(p) = 0\} . \quad (1)$$

Often times, one is interested in visualizing *isosurfaces* (or *isocontours* for 2D functions) which are functions of the form $f(p) = c$ for some constant c.

Different classes of implicit surfaces are of importance. Implicit surfaces that are defined by a polynomial are called *algebraic surfaces*. Since almost all continuous functions can be approximated by polynomials, algebraic surfaces represent a powerful class. Since polynomials have been studied and understood fairly well, one has a better mathematical handle at algebraic surfaces.

Another special class of implicit surfaces have density functions as their defining functions. Density functions decrease exponentially with increasing distance to a certain center point. The models where these functions find usage usually arise in the study of physical phenomena, e.g. atom models. Blinn [4] was one of the first to try to visualize these surfaces on a computer. They became to be known as Blinn's "*Blobby Model*" or *Metaballs*.

Yet another model which is gaining importance is the *discrete data set*. Such data set is commonly generated by various medical scanners such as MRI and CT. These data sets can also be seen as implicit functions, where $f(p) = v$ and v is the scanned value at the location p. We assume that if p is not on a grid point then some sort of interpolation from neighboring grid points defines v at p.

Rendering implicit surfaces is very difficult and computationaly very expensive. Usually existing algorithms to render implicit surfaces work only for special cases or special sets of functions. In fact, to date there is no such algorithm that can render arbitrary implicit surfaces accurately and in a stable and timely manner.

Algorithms for rendering 3D implicit surfaces can be classified into three groups: representation transforms, ray tracing, and surface tracking. The first approach tries to transform the problem from implicit surfaces to another representation which is easier to display such as polygon meshes [1][5][10][15] or parametric representations [16]. The second approach to rendering of implicit surfaces is to ray-trace them. Efficient ray-object intersection algorithms have been devised for algebraic surfaces [11], metaballs [4], and a class of functions for which Lipschitz constants can be efficiently computed [12][14]. The third approach to rendering implicit functions is to scan convert or discretize the function. Seder-

berg and Zundel [17] introduced such a scan line algorithm for a class of implicit functions that can be represented by Bernstein polynomials.

When it comes to drawing 2D implicit functions (curves) an attractive approach is to follow the path of the curve. The algorithm starts at a seed point on the curve. It then looks at the pixels adjacent to the seed point to see if any of them is the next pixel along the curve. Various algorithms exist which mainly differ in the method they use to find the next pixel along the curve.

One way to find the next pixel is based on *digital difference analysis* (DDA). A lot of work has been done in this field and the most popular algorithm applying this technique are Bresenham's and the midpoint line and circle drawing algorithms [8]. The rate of change in x and y is computed through fast integral updates of dx and dy, which are then used as decision variables to determine the next pixel center. This idea was generalized for algebraic curves by Hobby [13]. One problem of this approach is that the inherent integer arithmetic can only be exploited for a small set of algebraic functions. Furthermore, this algorithm is not able to deal properly with self intersecting curves. Hobby assumes that these intersection points are given by the user.

A different way to continue a curve would be to evaluate the underlying function at the center of the neighboring pixels to find the one with the smallest absolute function value. This idea has been implemented by Jordan et. al. [9] and was improved by Chandler [7]. Since there are only eight neighboring pixels, eight function evaluations are the maximum needed to find the next pixel on the curve. Since we will very likely be traveling in the same direction we have been traveling when we picked the current pixel, Chandler tests first the pixel continuing in the same direction, then its neighbors etc. For most 'nicely' behaving functions two to three function evaluations per pixel will be sufficient. To increase accuracy, Chandler prefers to look for a sign change rather than the smallest absolute value. Therefore, he evaluates the function not at pixel centers but rather halfway in between pixels. Although the algorithm does not depend on the actual description of the function and therefore is applicable to arbitrary functions, it is still not able to deal with self intersecting functions. To deal with such points the algorithm requires user intervention. Alternatively, it includes a brute-force Monte Carlo method to visualize the positive and negative regions of the function. Finally, if the curve enters and leaves the pixel in-

between the same two neighboring sample points, this algorithm will not be able to register a sign change and therefore fail.

For density functions, Arvo et. al. applied a numerical method known as the predictor-corrector method [2]. Their main goal is the rendering of iso-curves of medical data sets. In this paper we introduce an efficient and more robust predictor-corrector method. Our algorithm deals with self intersecting curves properly and increases the efficiency of the algorithm to less than one function evaluation per pixel on the average. Finally, unlike existing curve tracking methods, our algorithm can be applied to a large family of implicit functions including selfintersecting functions with at most two branches at the intersection point.

2. The Predictor-Corrector Method

The predictor-corrector method is a well known numerical algorithm which has its origin in solving ordinary differential equations and bifurcation theory. Arvo and Novins [2] used a two dimensional standard method for extracting isocurves in a 2D medical image. Although the predictor-corrector method offers ways to detect and properly deal with self intersecting curves and curves “very close” to themselves, Arvo and Novins did not explore these capabilities.

The predictor-corrector method is a numerical continuation method (tracking algorithm). The goal is to find a point set which represents a discretized approximation of a curve that is defined by some implicit function as in Equation (1). As the name suggests, the predictor-corrector method consists of two steps. In the predictor step we take a (more or less sophisticated) guess of a point on the curve, denoted by p_p. Depending on how much thought and work we put into predicting the point p_p, it will be closer or farther away from an actual point on the curve. Depending on the assumption we make about the curve and its actual shape, we will not be able to always accurately predict an exact point on the curve. In the corrector step we use our predicted point p_p as a starting point for an iterative method that converges to an actual point on the curve. We denote this new, corrected point, by p_c. We repeat these two steps until we tracked the whole curve. The resulting discrete point set can then be displayed on a screen.

There are many different ways to implement the predictor or corrector steps. We use the standard Euler-Newton method with an adaptive step size and introduce techniques to handle selfintersecting curves.

2.1 The Predictor Step

The focus of the predictor step is to find a point p_p that is very close or even on the curve, so that the costs for the corrector step are reduced or even eliminated. If we already know points on the curve, we can approximate the curve through a polynomial and get a new approximate for this curve through extrapolation or interpolation.

Finding the very first point on the curve is not always easy. One usually has some understanding of the kind of function one tries to discretize and can therefore provide an initial seed value to the algorithm. If this is not possible, one needs to apply a brute force algorithm or some other heuristic to find an initial seed value. The image space is limited (usually a screen in computer graphics). Therefore one rather brute-force method would be to impose a grid onto the screen and evaluate the underlying implicit function at each grid point to choose a closest point (where $|f(p)|$ becomes smallest). Another possibility would be to look for a sign change of the underlying function between neighboring grid points. We employ a faster method that is based on a quadtree subdivision algorithm as explained in Section 3.1.

We now present one way to implement the predictor step. *One-step methods* only employ one point (the point of the previous iteration) to predict a new point. The standard Euler method is a good example of a one-step method. However, one can envision the use of Hermites method which employs more than one point and is therefore called *multi-step method*. Although multi-step methods are usually more expensive, they lead to better, higher order approximations, reducing the number of necessary corrector steps.

Euler's method starts at a point which is known to be on the curve. Because this is a one-step method, each point on the curve will lead to exactly one new point on the curve. Since we execute many predictor-corrector iterations to obtain a point set that approximates our curve, it is just obvious to use the resulting point of the previous iteration as the initial point for the current iteration. We call this initial point $p_i = (x_i, y_i)$. We also want to have some control over the distance between the new point and

the initial point. That is, we want p_p to be located at a certain distance Δs from p_i and not at some unpredictable "end" of the curve. Δs is called the arc length parameter and it represents a parameterization of our curve. The linear approximation of the predicted point results in

$$\begin{aligned} x_p &= x_i + dx\Delta s \\ y_p &= y_i + dy\Delta s \end{aligned}, \tag{2}$$

where $p_p = (x_p, y_p)$ now depends on the arc length and the initial point. Since Δs is the euclidean distance between p_p and p_i, we need to make sure that:

$$dx^2 + dy^2 = 1 \tag{3}$$

Equation (3) defines a unit circle around the origin. Combining it with Equation (2) we are describing a circle of radius Δs and origin p_i. Next, we have to find the intersection points of that circle with our curve. We choose one of these points as our new predicted value. Since the origin of this circle lies already on the curve, we are guaranteed that our circle will intersect our implicit curve, unless the whole curve is contained within the circle. In that particular case, the choice of the distance Δs between neighboring points is not appropriate and needs to be changed. If Δs is in the order of the size of a pixel and the curve is still contained within such a small circle, the discretization of this curve would just be one pixel and we are done.

Assuming a closed curve, we will actually get an even number of intersection points. Choosing our step size (arc length) small enough, we will get exactly two intersection points (see Figure 1). However, in the case of a self intersecting curve or when a curve comes "close" to itself, we will get more than two intersection points, even if Δs is very small. We will explore these special cases in Section 2.2.

The second condition for the new predicted point is, of course, that it should lie on the curve defined by the underlying implicit function *f*. That means we require

$$f(x_p, y_p) = 0\,. \tag{4}$$

Since we are processing an arbitrary functions *f* we are not able to use Equation (4) directly to compute p_p. Instead, we substitute Equation (2) into Equation (4) and through a Taylor series expansion we are

able to compute an approximation to the actual point on the curve. This approximation serves as the predicted value p_p.

Let's have a look at the equations. The substitution results in

$$f(x_i + dx\Delta s, y_i + dy\Delta s) = 0. \tag{5}$$

Now the actual approximation takes place. We expand Equation (5) in a Taylor series and keep only the linear terms. That results in:

$$f(x_i, y_i) + \frac{\partial}{\partial x} f(x_i, y_i) dx\Delta s + \frac{\partial}{\partial y} f(x_i, y_i) dy\Delta s = 0 \tag{6}$$

The initial point was assumed to be on the curve, i.e. $f(x_i, y_i) = 0$. We conclude

$$\frac{\partial}{\partial x} f(x_i, y_i) dx\Delta s + \frac{\partial}{\partial y} f(x_i, y_i) dy\Delta s = 0 \tag{7}$$

Using Equation (3) and Equation (7), we can actually solve for the unknowns (dx, dy). Using a short-cut notation for the partial derivatives of $f(x_i, y_i)$, where $\frac{\partial}{\partial x} f(x_i, y_i)$ is replaced by f_x (and similarly for the partial derivative in y) the result can be written as

$$\begin{aligned} dx &= \delta \frac{f_y}{\sqrt{f_x^2 + f_y^2}} \\ dy &= -\delta \frac{f_x}{\sqrt{f_x^2 + f_y^2}} \end{aligned} \tag{8}$$

which is also proven by Baker and Overman [3]. Here δ stands for the sign plus or minus which represents the direction in which we continue to track the curve. In the case of the seed point we want to make sure, that we trace both directions. However, in the general case we don't want to step back into the direction we were coming from. A simple sign check of the scalar product of the direction, where we were coming from and our new direction, helps us determining the appropriate δ.

Although our new predicted value lies on a circle around the initial point, we only find a point near the curve because of the approximation in Equation (6). If we take a closer look at Equation (7), the one we actually used as an approximation to Equation (5), we realize that (dx, dy) describes the components

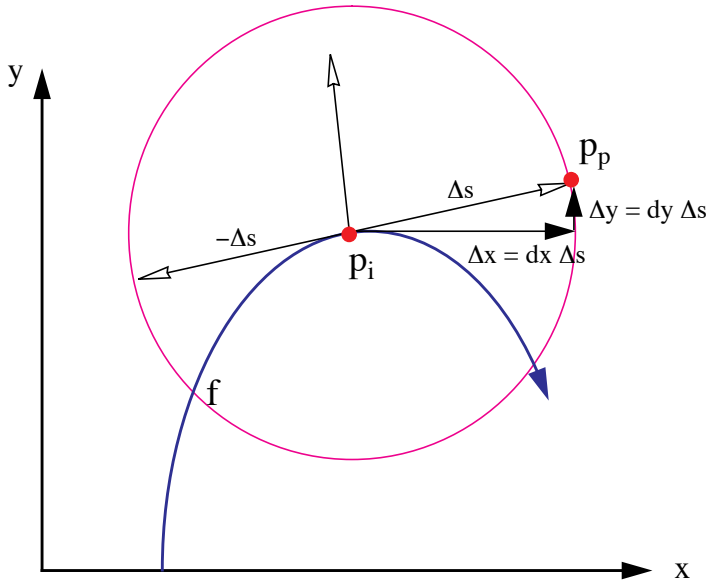

FIGURE 1. The predictor step. Here Δs is the step size we want to step along the curve starting at the initial point p_i. Because Equation (7) is only an approximation, the predicted value will not necessarily lie on the curve described by f.

of the tangent to the curve at the initial point. That leads us to a better geometric understanding of this process, captured in Figure 1.

2.2 Bifurcation Points

The intersection of the circle equation (originating in Equation (3)) with the implicit curve results usually in two points. As pointed out earlier, in case of a selfintersecting curve (Figure 2a) or if it just comes "close" to itself (Figure 2b) we expect at least four intersection points. These special points are also called *bifurcation points*. Looking at our formulas in Equation (7) and Equation (3), we are only able to compute two points, since it is an intersection of a quadratic (Equation (3)) curve with a linear curve (Equation (7)). In a case of a bifurcation point the linear approximation of the Taylor series as in Equation (7) will not be enough. In fact, from the implicit function theorem we can conclude, that the first partial derivatives will be zero for these bifurcation points. Therefore we cannot compute (dx, dy) as derived above. We have to include the second order terms of the Taylor series expansion in Equation (6) to get the appropriate results.

The more accurate approximation to Equation (5), replacing Equation (6), is now

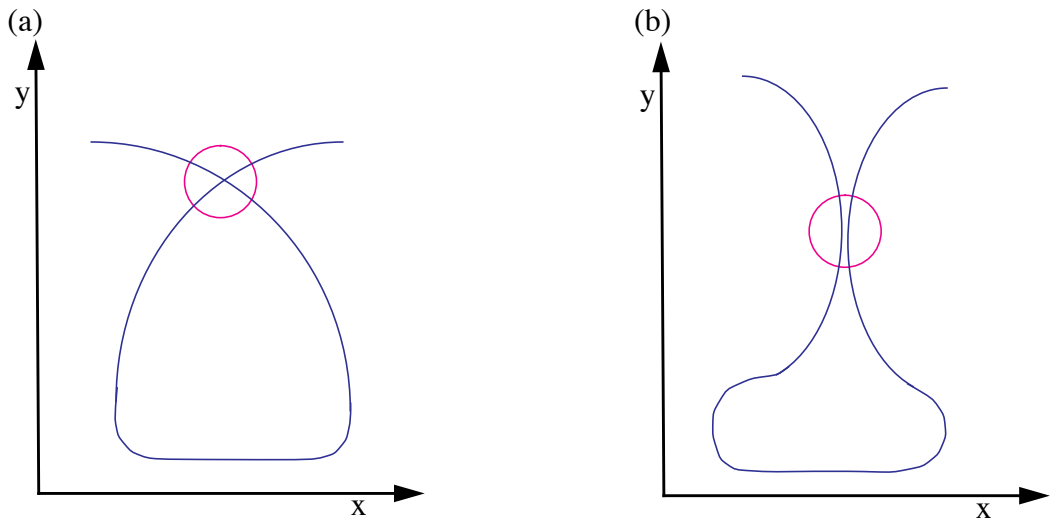

FIGURE 2. Possible cases of bifurcation points where the circle represents the actual step size. (a) The curve intersects itself; (b) The curve comes very “close” to itself.

$$\begin{aligned} &f(x_i, y_i) + f_x(x_i, y_i)dx\Delta s + f_y(x_i, y_i)dy\Delta s + \\ &f_{xx}(x_i, y_i)dx^2\Delta s^2 + 2f_{xy}(x_i, y_i)dxdy\Delta s^2 + f_{yy}(x_i, y_i)dy^2\Delta s^2 = 0 \end{aligned} \tag{9}$$

As already mentioned, the first three terms in the sum vanish, which leads to

$$f_{xx}(x_i, y_i)dx^2\Delta s^2 + 2f_{xy}(x_i, y_i)dxdy\Delta s^2 + f_{yy}(x_i, y_i)dy^2\Delta s^2 = 0 \tag{10}$$

In order to increase the readability of the following formulas, we use the following notation:

$$\begin{aligned} A &= f_{xx}(x_i, y_i) \\ B &= f_{xy}(x_i, y_i) \\ C &= f_{yy}(x_i, y_i) \end{aligned} \tag{11}$$

If we consider Equation (10) as a quadratic equation in the quotient q defined by dx/dy, we get the following solution:

$$q_{1,2} = \frac{dx}{dy} = \frac{-B \pm \sqrt{B^2 - AC}}{A} = \frac{C}{-B \mp \sqrt{B^2 - AC}} \tag{12}$$

To actually compute dx and dy, we substitute this result in Equation (3), which results in

$$\begin{pmatrix} dx \\ dy \end{pmatrix} = \pm \begin{pmatrix} \sqrt{\dfrac{1}{\dfrac{1}{q_{1,2}^2}+1}} \\ \sqrt{\dfrac{1}{q_{1,2}^2+1}} \end{pmatrix} = \pm \begin{pmatrix} \sqrt{\dfrac{C^2 - AC + 2B\,(B \pm \sqrt{B^2 - AC})}{(A-C)^2 + 4B^2}} \\ \sqrt{\dfrac{A^2 - AC + 2B\,(B \mp \sqrt{B^2 - AC})}{(A-C)^2 + 4B^2}} \end{pmatrix} \tag{13}$$

where any arbitrary combination of plus and minus from Equation (12) and Equation (13) results in four different distinct direction vectors (dx, dy). We will store all four directions in a list and follow their branches one at a time. This is explained in greater detail in Section 3.

We discussed the usual case of two intersection points and studied in detail what happens when these formulas fail when we encounter bifurcation points. Unfortunately, it could happen that we do not have four intersection points, but six, eight, or more. That means, that actually more then the two branches, as shown in Figure 1, meet at the same area. In that case, we realize that the second partial derivatives in Equation (9) vanishes also, leaving us with no choice other than including even higher order terms in the approximation of the Taylor series. This implies that we will have to solve higher order polynomials to produce the expected number of solutions.

2.3 The Corrector Step

Although we have tried hard to predict a new point p_p on the curve, it is not guaranteed to fall accurately on the curve. We can only hope to find a point that is "very close" to the curve. The goal of the *corrector step* is to find a "closest" point on the curve, starting at p_p. Among the many methods for finding zero points of functions, the Newton method is very stable and fast, and therefore very common. The Newton method is a numerical iteration method which converges to a zero point p^* of a function f, starting at a point p_0 that should be fairly close to the zero point already. In our case, the starting point for the Newton method will be p_p – the result of the predictor step.

The predictor step produced an initial value $p_0 = p_p = (x_p, y_p) = (x_0, y_0)$ for the iteration method. At each iteration step we take the new and improved value from the previous iteration and try to improve it even further, so that we actually end up on the curve itself. The result of the current iteration can be written as:

$$p_{k+1} = \begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k + \Delta x \\ y_k + \Delta y \end{pmatrix}, \tag{14}$$

where k marks the iteration number.

But how can we determine $(\Delta x, \Delta y)$? Here we exploit the same idea as we did in Euler's Method. Of course, we want that our new point (x_{k+1}, y_{k+1}) lies on the curve. So we add the condition that p_{k+1} is a curve point:

$$f(x_{k+1}, y_{k+1}) = 0. \tag{15}$$

The same ideas and manipulations as used for Equation (4) will lead us towards a solution. Substituting Equation (14) into Equation (15) and using a Taylor series expansion of f will result in:

$$f_x(x_k, y_k)\Delta x + f_y(x_k, y_k)\Delta y = -f(x_k, y_k). \tag{16}$$

That is, in fact, the formula for Newton's method for a two dimensional function. Since this is one equation with two unknowns, we have the freedom and duty to specify another condition, we want our iterate to satisfy. First of all, we want to make sure that our iteration process converges. Other than that, it would be nice if we could guarantee some sort of constant discretization of our curve. That is, we do not want the distance between samples to vary too much. Both of these ideas can be realized if we restrict our convergence to move in a right angle away from our previous iterate. To be more specific, the triangle formed by the points p_i, p_k and p_{k+1} should form a right angled triangle. Here, p_i is the initial point on the curve – the one we used in the predictor step to compute the point p_p. p_k and p_{k+1} are the previous and current iterates. In Figure 1 it was shown that this condition on our new iterate results in a somewhat circular movement around the initial point p_i. Because of this circular movement, we stay in a certain proximity to p_i, which increases the chances of convergence.

More formally, we want to add the condition that the vector $\overrightarrow{p_k p_i}$ is perpendicular to $\overrightarrow{p_k p_{k+1}}$. That is expressed in the following manner:

$$(p_i - p_k) \bullet (p_{k+1} - p_k) = \begin{bmatrix} x_i - x_k \\ y_i - y_k \end{bmatrix}^T \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \Delta x\,(x_i - x_k) + \Delta y\,(y_i - y_k) = 0 \tag{17}$$

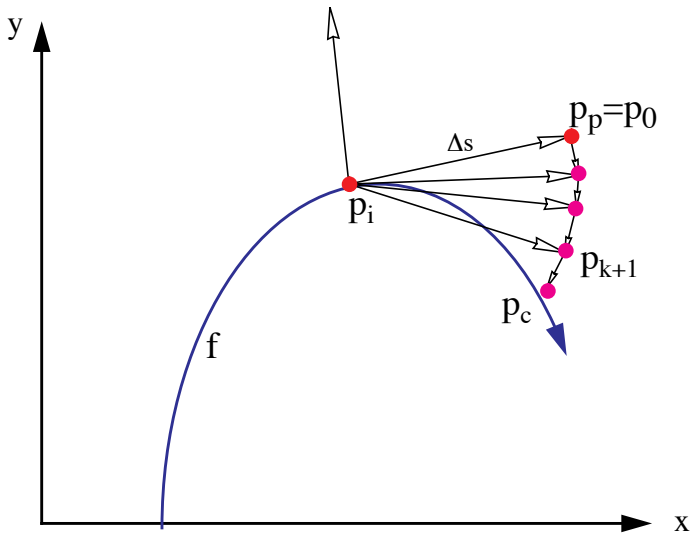

FIGURE 3. The corrector step. The Newton iterations and a forced circular movement of the iterates p_{k+1} converge towards an actual point on the curve p_c.

Equation (16) together with Equation (17) uniquely describe a new, corrected iterate p_{k+1}, which concludes the Corrector step. A geometric interpretation of these steps is illustrated in Figure 3.

3. The Algorithm

The pseudo code for our numerical algorithm is given in Figure 4. The algorithm starts by looking for a seed point (using a space subdivision method, as described in Section 3.1). The initial point will be most likely on some arbitrary part of the curve. Therefore, we will usually have two possible directions we can follow. We record both in a list for later retrieval (Line 3). At a bifurcation point a pixel will have four branches leaving it. In this case we will also record all possible branch directions in this list (Line 18). After retrieving a starting point for a new branch (Line 5) we trace the branch until we find another bifurcation point or until we reach the limits of our display device. In the case of a closed curve we also need to check when we get back to an already traced branch of the curve. A simple way to do this is explained in Section 3.5.

```
1. find an initial point p_i on the surface;
2. determine initial step size Δs;
3. record (p, Δs, bifurlist);                    /* for both initial direction */
4. repeat
5.     get p_i, Δs from bifurlist
6.     repeat                                    /* continue current branch */
7.         repeat                                /* find a reliable new point on the curve */
8.             p_p := Predictor(p_i)
9.             p_c := Corrector(p_p)
10.            C := approx_curvature(p_c, p_p)
11.            Δs := Δs/2
12.        until (p_c converges AND 1/(2C) < Δs) OR bifur_test(p_c)
13.        Δs := Δs*2;
14.        if 1/(2C) >Δs then
15.            increase Δs;
16.        draw_line(p_c, p_i);
17.        if bifur_test(p_c) then
18.            record(p_c, Δs, bifurlist)
19.        p_i := p_c;
20.    until (p_c is in the covered region or outside the screen OR bifur_test(p_c));
21. until bifurlist is empty
```

FIGURE 4. pseudo code for the curve tracking algorithm.

For each new predictor point we need to make sure that the corrector didn't fail (Line 7 through Line 12). Once we have gained confidence about our corrected point, we check whether we can adjust the step size (Line 15). We connect the previous point to the current point by a simple line (Line 16). If we found a new bifurcation point, we need to record it (Line 18) and quit this loop. In any other case we repeat this procedure and find another point on the same branch.

We now describe in more detail the implementation of some of the more involved segments of the algorithm.

3.1 Finding a Starting Point

If a seed point is not provided by the user, we employ a space subdivision approach. We divide the image region into four quadrants and compute the function at each corner of the quadrants. If the sign of the function is the same at all corners, we will apply the same subdivision to all four quadrants in a breath-first scheme. If we do find a sign change, we continue to subdivide this specific quadrant in a depth-first scheme up to the pixel- or even subpixel level, depending on how accurate our initial point should be.

3.2 Implementing the Corrector Step

Since the predicted point is only an approximation of a curve point, the corrector step is employed to converge to an actual point on the curve. Because of the approximations in our formulas, the Newton corrector step (explained in Section 2.3) usually needs to be repeated a few times. That is done within the Corrector procedure in Line 9. But how many times should we repeat the Newton step? Of course, we have to fight round off errors caused by the discrete computer arithmetic as well. The quality of our discretized curve greatly depends on our stopping criteria, as well as the successful completion of our algorithm. Therefore, we need to know when $f(p_{k+1})$ gets close enough to zero. The problem is that *close* means very different things for different functions. If we had more knowledge about the function, we could employ an absolute error test, like $f(p_{k+1}) < \varepsilon$. For a general algorithm, we will have no idea how small ε needs to be to assure spatial proximity. This notion of *being close* to the curve might also change at different ranges for the curve.

The method we chose adapts the value of ε and utilizes the fact that we draw to a finite grid. Every once in a while (e.g. every tenth time we call the Corrector procedure) we check the quality of the corrected point p_c. For this quality check we define a rectangular region centered at p_c and check for a sign change of the function f at the corner points. The size of this rectangular region can be a parameter L specified by the user or simply the current grid size. If there is a sign change in the corner points, we know that our curve will be at most $L/\sqrt{2}$ away. Therefore, we could relax our stopping criteria and set ε to $1.1|f(p_{k+1})|$. On the other hand, if we find that our corrected value isn't close enough to our curve, we should strengthen our stopping condition and set ε to $10^{-2}|f(p_{k+1})|$ and redo the corrector

step. This finally results in a relative (adaptive) error test, which is certainly superior than an absolute (static) error test.

3.3 Determining The Step Size Δs

The other crucial and very sensitive parameter of this algorithm is the step size Δs . One big advantage of our algorithm is the freedom to choose the step size. When the curve shows a small curvature we will choose a fairly large Δs , allowing the algorithm to step quickly through this “almost linear” part of the curve. On the other hand, when the curve changes rather quickly, i.e. the curvature of the function is high, we have to decrease Δs appropriately. This adaptive mechanism gives us the ability to capture the nature of our curve in greater detail when necessary.

The mathematical definition of the curvature is

$$C(p) = \lim_{q \to p} \frac{\alpha(p) - \alpha(q)}{\widehat{pq}}, \tag{18}$$

where $\alpha(p)$ denotes the angel of the tangent with the x-axis and $\widehat{pq}$ denotes the arc length between the two points. For implicit functions, C can be computed as

$$C = \frac{-f_y^2 f_{xx} + 2 f_x f_y f_{xy} - f_x^2 f_{yy}}{\left(f_x^2 + f_y^2\right)^{3/2}} \tag{19}$$

(See [6]). We compute first derivatives anyway for the Newton corrector, but would like to avoid computing second derivatives. Therefore, we use an approximation to the curvature that is based on Equation (18), where we use the current and previous point on the curve. Since we have the first derivatives already computed we can precisely compute the tangent angles. The arc length between the two points is simply approximated with the current step size. To compute the initial step size we could either use Equation (19) or just use the grid size. Choosing the initial step size for a bifurcation point can be a problem. If we are stepping too far, we might ‘loose track’ of the current branch. If we are stepping too close, the corrector might converge to a different, neighboring branch. Therefore we left it up to the user to choose this initial step size.

Another indicator of the quality of our step size would be the number of corrector steps we had to take before we converged to a curve point. In our predictor step we are basically following the tangent of the curve. If the curve is fairly linear the predicted point p_p will be almost on the curve, resulting in just very few corrector steps. If the curve changes its direction rather fast, p_p will not be the best guess and we expect more correction steps. Therefore, if we use at most one Newton step to find a point on the curve, we simply double the step size.

3.4 Detecting Bifurcation Points

Bifurcation points are points where the curve intersects itself. At these points the first derivatives of the function evaluate to zero. Of course, it will be very unlikely to find the mathematically exact bifurcation point (in case one exists), because of the numerical nature of the algorithm. Therefore, we need to keep watching the first derivatives of a function for regions where they are *close* to zero. We have seen already how difficult such a test can be (see Section 3.2). Therefore, whenever we find a sign change in one of the partial derivatives (from the previous to the current point) we have good reason to suspect that there might be a bifurcation point close by. As a result, we decrease our step size and repeat the last predictor/corrector step until the step size is smaller than a given size, which can be defined by the user. If there is still reason to believe that there might be a bifurcation point, we create a rectangular region around the current point on the curve. Similar to the test we performed in Section 3.2, we test whether there is a sign change in the partial derivatives f_x and f_y. If we find a sign change in both partial derivatives we conclude that we detected a bifurcation point. Otherwise, we conclude that we probably found a turning point of the curve.

3.5 Stopping Criteria

We assume that we deal with a closed curve. Therefore, we need to find out if and when we returned to a part of a branch we have traced already. If we do not include this exit test (in Line 20), we will trace the curve over and over again, ending up in an infinite loop.

Because of the nature of a numeric algorithm it is very unlikely to return to the exact same point from where we started. Therefore, we cannot just test for equality. We also do not want to test the proximity

of the new point p_c to every one of the computed values along the curve. That might be possible for the first few values, when the cost is still low, but certainly not for arbitrary curves, where we have to compute many points to reasonably approximate the curves shape. The other problem is again the meaning of "very close". To solve this problem, we maintain a counter that counts the number of times we attempt to draw a pixel that has been drawn already. When the algorithm computes a point on the curve that is mapped to a pixel that was not drawn before, we set the counter to zero. Every time we compute a new point on the curve within a pixel that was previously drawn, we increase a counter by one. If the counter's value exceeds some threshold, the algorithm stops pursuing this path.

TABLE 1. Performance results comparing our algorithm. The empty cells denote cases where Chandler's algorithm could not draw the curve.

Figure	# Function evaluation per pixel		# pixels per predictor	# pixels per predictor
	Our Method	Chandler		
5a	1.89	2.68	6.21	2.55
5b	2.01	2.80	6.79	3.20
5c	3.27	2.92	3.51	2.49
6a	2.88	–	6.66	5.04
6b	4.18	–	4.05	3.67
6c	4.55	–	2.38	2.30

4. Results

We applied our algorithm to many different functions with and without bifurcation points and found that they were drawn correctly in any image resolution of 128^2 to 512^2. For images of low resolution (32^2 and sometimes 64^2) the user may have to explicitly specify a smaller step size. We have chosen three very common functions without a bifurcation point (Figure 5) and three interesting functions with many bifurcation points (Figure 6). Some interesting statistics for each group of functions is summarized in Table 1 and compared to Chandlers algorithm [7]. The test cases where drawn onto a 5122 image. Drawing times, for all examples, was under 0.2 seconds on a Silicon Graphics Crimson (150Mhz R4400). In the table we also provide the average number of pixels per one predictor step (showing the adaptive step size), and the average number of correct steps per predictor.

In general, the number of function evaluations per pixel for low resolutions is expectably high, since the curvature of a curve considered within a pixel is larger than if the same curve is subdivided into smaller parts (i.e. for smaller pixel sizes). Therefore, the step size (in terms of the number of pixels) is smaller and the algorithm must perform more calculations on a per pixel basis. For high resolutions we have much fewer function evaluations per pixel than Chandler's algorithm without loss of quality. This demonstrates the power of the adaptive step size.

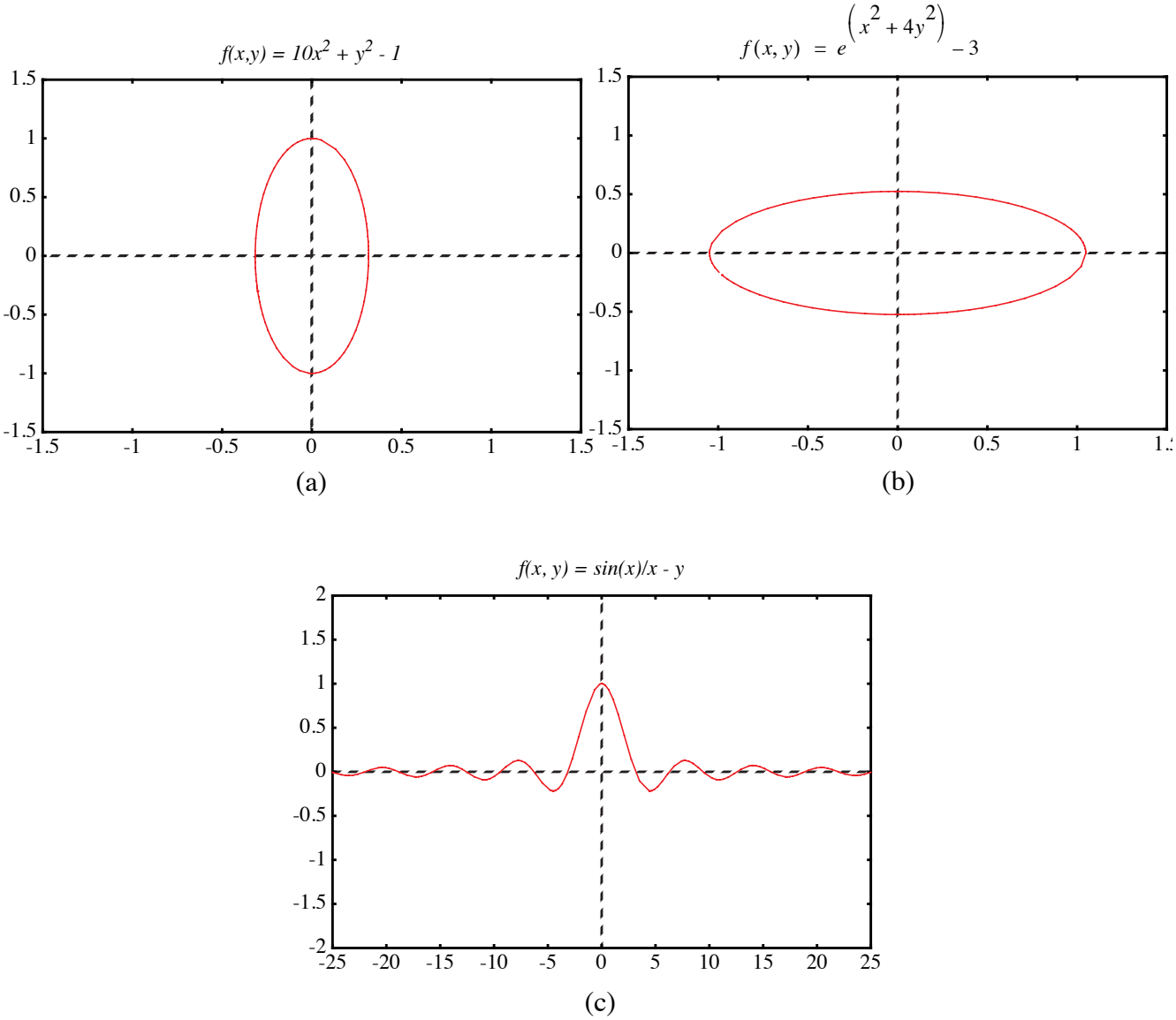

FIGURE 5. Curves without bifurcation points.

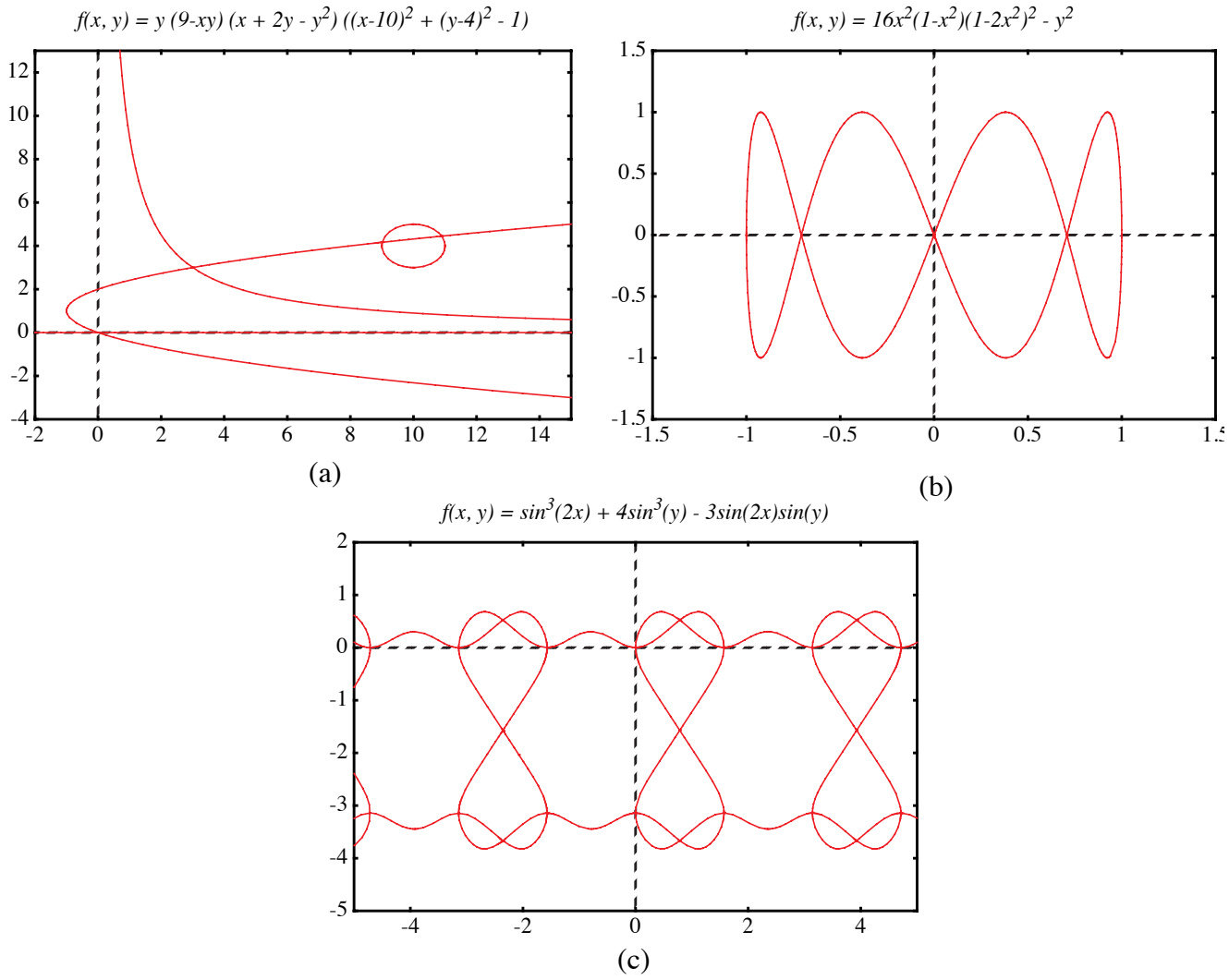

FIGURE 6. Curves with bifurcation points.

5. Conclusions

We presented here a numerical algorithm that is able to draw a large family of implicit functions. Through an adaptive step size design our performance is greatly optimized. We have shown that our approach outperforms previous tracking algorithms.

Our future research includes the improvement of the accuracy of the predictor step. Since we already know the derivatives at the previous point and, most of the time, also have this information available for the point before that, we could use the Hermite interpolation scheme and approximate our curve with a

polynom of third degree. This higher order approximation of our curve should result in fewer corrector steps. Since the higher order polynomial is computed only at the predictor step, saving several corrector steps, we expect a more efficient algorithm. A similar improvement can be made when connecting a new point on the curve with the previous one. Instead of a line through these point, one could use a higher order polynomial. Improvements to the accuracy of the corrector procedure are also possible, for example, by using a better corrector than Newton's. Finally, we plan to explore the extension of our methods to 3D voxelization of implicit surfaces.

6. References

[1] Allgower E. and S. Gnutzmann, "Simplicial Pivoting for Mesh Generation of Implicitly Defined Surfaces", Computer Aided Geometric Design, 8(4), 1991, pp. 30.

[2] Arvo J. and K. Novins, "Iso-Contour Volume Rendering", Proceedings of 1994 Symposium On Volume Visualization.

[3] Baker G. and E. Overman, "The Art of Scientific Computing", Draft Edition, The Ohio State Bookstore.

[4] Blinn J.F., "A Generalization of Algebraic Surface Drawing", ACM Transactions on Graphics, 1(3), 1982, pp. 235-256.

[5] Bloomenthal J., "Polygonization of Implicit Surfaces", Computer Aided Geometric Design, 5(4), 1988, pp. 341-356.

[6] Bronstein I. N. and Semendjajew, K. A. "Taschenbuch der Mathematik", BSB B.G.Teubner Verlagsgesellschaft, Leipzig, 1985.

[7] Chandler R.E., "A Tracking Algorithm for Implicitly Defined Curves", IEEE Computer Graphics and Applications, 8(2), March 1988, pp. 83-89.

[8] Foley J. D., A. van Dam, S. K. Feiner and J. F. Hughes, "Computer Graphics Principles and Practice", Addison-Wesley, 1990.

[9] Jordan B.W., W.J. Lennon, and B.D. Holm, "An Improved Algorithm for the Generation of Nonparametric Curves", IEEE Trans. Computers, Vol. C-22, 1973, pp. 1052-1060.

[10] Hall M. and J.Warren, "Adaptive Polygonization of Implicitly Defined Surfaces", IEEE Computer Graphics and Applications, 10(6), November 1990, pp. 33-42.

[11] Hanrahan P., "Ray Tracing Algebraic Surfaces", Computer Graphics (SIGGRAPH'83 Proceedings), 17(3), 1983, pp. 83-90.

[12] J.C. Hart. Ray Tracing Implicit Surfaces. WSU Technical Report EECS-93-014. Chapter in "Design, Visualization, and Animation of Implicit Surfaces," ACM SIGGRAPH '93 intermediate course notes, J. Bloomenthal and B. Wyvill, eds., Aug. 1993.

[13] Hobby J.D., "Rasterization of Nonparametric Curves", ACM Transactions on Graphics, 9(3), July 1990, pp. 262-277.

[14] Kalra D. and A.H. Barr, "Guaranteed Ray Intersections with Implicit Surfaces", Computer Graphics (SIGGRAPH'89 Proceedings), 23, 1989, pp. 297-306.

[15] Lorensen W. and H. Cline, “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”, Computer Graphics, 21(4), 1987, pp. 163-169.

[16] Sederberg T.W., D.C.Anderson, and R.N.Goldman, “Implicitization, Inversion, and Intersection of Planar Rational Cubic Curves”, Computer Vision, Graphics, and Image Processing, 31(1), 1985, pp. 89-102.

[17] Sederberg T.W. and A.K.Zundel, “Scan Line Display of Algebraic Surfaces”, Computer Graphics (SIGGRAPH’89 Proceedings), 23(4), 1989, pp. 147-156.

Efficient Rasterization of Implicit Functions

Torsten Möller and Roni Yagel

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio

{moeller, yagel}@cis.ohio-state.edu

Abstract

Implicit curves are widely used in computer graphics because of their powerful features for modeling and their ability for general function description. The most popular rasterization techniques for implicit curves are space subdivision and curve tracking. In this paper we are introducing an efficient curve tracking algorithm that is also more robust then existing methods. We employ the Predictor-Corrector Method on the implicit function to get a very accurate curve approximation in a short time. Speedup is achieved by adapting the step size to the curvature. In addition, we provide mechanisms to detect and properly handle bifurcation points, where the curve intersects itself. Finally, the algorithm allows the user to trade-off accuracy for speed and vice a versa. We conclude by providing examples that demonstrate the capabilities of our algorithm.

1. Introduction

Implicit surfaces are surfaces that are defined by implicit functions. An implicit function f is a mapping from an n dimensional space R^n to the space R, described by an expression in which all variables appear on one side of the equation. This is a very general way of describing a mapping. An n dimensional implicit surface S is defined by all the points p in R^n such that the implicit mapping f projects p to 0. More formally, one can describe S by

$$S_f = \{p \mid p \in R^n, f(p) = 0\} . \quad (1)$$

Often times, one is interested in visualizing *isosurfaces* (or *isocontours* for 2D functions) which are functions of the form $f(p) = c$ for some constant c.

Different classes of implicit surfaces are of importance. Implicit surfaces that are defined by a polynomial are called *algebraic surfaces*. Since almost all continuous functions can be approximated by polynomials, algebraic surfaces represent a powerful class. Since polynomials have been studied and understood fairly well, one has a better mathematical handle at algebraic surfaces.

Another special class of implicit surfaces have density functions as their defining functions. Density functions decrease exponentially with increasing distance to a certain center point. The models where these functions find usage usually arise in the study of physical phenomena, e.g. atom models. Blinn [4] was one of the first to try to visualize these surfaces on a computer. They became to be known as Blinn's "*Blobby Model*" or *Metaballs*.

Yet another model which is gaining importance is the *discrete data set*. Such data set is commonly generated by various medical scanners such as MRI and CT. These data sets can also be seen as implicit functions, where $f(p) = v$ and v is the scanned value at the location p. We assume that if p is not on a grid point then some sort of interpolation from neighboring grid points defines v at p.

Rendering implicit surfaces is very difficult and computationaly very expensive. Usually existing algorithms to render implicit surfaces work only for special cases or special sets of functions. In fact, to date there is no such algorithm that can render arbitrary implicit surfaces accurately and in a stable and timely manner.

Algorithms for rendering 3D implicit surfaces can be classified into three groups: representation transforms, ray tracing, and surface tracking. The first approach tries to transform the problem from implicit surfaces to another representation which is easier to display such as polygon meshes [1][5][10][15] or parametric representations [16]. The second approach to rendering of implicit surfaces is to ray-trace them. Efficient ray-object intersection algorithms have been devised for algebraic surfaces [11], metaballs [4], and a class of functions for which Lipschitz constants can be efficiently computed [12][14]. The third approach to rendering implicit functions is to scan convert or discretize the function. Seder-

berg and Zundel [17] introduced such a scan line algorithm for a class of implicit functions that can be represented by Bernstein polynomials.

When it comes to drawing 2D implicit functions (curves) an attractive approach is to follow the path of the curve. The algorithm starts at a seed point on the curve. It then looks at the pixels adjacent to the seed point to see if any of them is the next pixel along the curve. Various algorithms exist which mainly differ in the method they use to find the next pixel along the curve.

One way to find the next pixel is based on *digital difference analysis* (DDA). A lot of work has been done in this field and the most popular algorithm applying this technique are Bresenham's and the midpoint line and circle drawing algorithms [8]. The rate of change in x and y is computed through fast integral updates of dx and dy, which are then used as decision variables to determine the next pixel center. This idea was generalized for algebraic curves by Hobby [13]. One problem of this approach is that the inherent integer arithmetic can only be exploited for a small set of algebraic functions. Furthermore, this algorithm is not able to deal properly with self intersecting curves. Hobby assumes that these intersection points are given by the user.

A different way to continue a curve would be to evaluate the underlying function at the center of the neighboring pixels to find the one with the smallest absolute function value. This idea has been implemented by Jordan et. al. [9] and was improved by Chandler [7]. Since there are only eight neighboring pixels, eight function evaluations are the maximum needed to find the next pixel on the curve. Since we will very likely be traveling in the same direction we have been traveling when we picked the current pixel, Chandler tests first the pixel continuing in the same direction, then its neighbors etc. For most 'nicely' behaving functions two to three function evaluations per pixel will be sufficient. To increase accuracy, Chandler prefers to look for a sign change rather than the smallest absolute value. Therefore, he evaluates the function not at pixel centers but rather halfway in between pixels. Although the algorithm does not depend on the actual description of the function and therefore is applicable to arbitrary functions, it is still not able to deal with self intersecting functions. To deal with such points the algorithm requires user intervention. Alternatively, it includes a brute-force Monte Carlo method to visualize the positive and negative regions of the function. Finally, if the curve enters and leaves the pixel in-

between the same two neighboring sample points, this algorithm will not be able to register a sign change and therefore fail.

For density functions, Arvo et. al. applied a numerical method known as the predictor-corrector method [2]. Their main goal is the rendering of iso-curves of medical data sets. In this paper we introduce an efficient and more robust predictor-corrector method. Our algorithm deals with self intersecting curves properly and increases the efficiency of the algorithm to less than one function evaluation per pixel on the average. Finally, unlike existing curve tracking methods, our algorithm can be applied to a large family of implicit functions including selfintersecting functions with at most two branches at the intersection point.

2. The Predictor-Corrector Method

The predictor-corrector method is a well known numerical algorithm which has its origin in solving ordinary differential equations and bifurcation theory. Arvo and Novins [2] used a two dimensional standard method for extracting isocurves in a 2D medical image. Although the predictor-corrector method offers ways to detect and properly deal with self intersecting curves and curves “very close” to themselves, Arvo and Novins did not explore these capabilities.

The predictor-corrector method is a numerical continuation method (tracking algorithm). The goal is to find a point set which represents a discretized approximation of a curve that is defined by some implicit function as in Equation (1). As the name suggests, the predictor-corrector method consists of two steps. In the predictor step we take a (more or less sophisticated) guess of a point on the curve, denoted by p_p. Depending on how much thought and work we put into predicting the point p_p, it will be closer or farther away from an actual point on the curve. Depending on the assumption we make about the curve and its actual shape, we will not be able to always accurately predict an exact point on the curve. In the corrector step we use our predicted point p_p as a starting point for an iterative method that converges to an actual point on the curve. We denote this new, corrected point, by p_c. We repeat these two steps until we tracked the whole curve. The resulting discrete point set can then be displayed on a screen.

There are many different ways to implement the predictor or corrector steps. We use the standard Euler-Newton method with an adaptive step size and introduce techniques to handle selfintersecting curves.

2.1 The Predictor Step

The focus of the predictor step is to find a point p_p that is very close or even on the curve, so that the costs for the corrector step are reduced or even eliminated. If we already know points on the curve, we can approximate the curve through a polynomial and get a new approximate for this curve through extrapolation or interpolation.

Finding the very first point on the curve is not always easy. One usually has some understanding of the kind of function one tries to discretize and can therefore provide an initial seed value to the algorithm. If this is not possible, one needs to apply a brute force algorithm or some other heuristic to find an initial seed value. The image space is limited (usually a screen in computer graphics). Therefore one rather brute-force method would be to impose a grid onto the screen and evaluate the underlying implicit function at each grid point to choose a closest point (where $|f(p)|$ becomes smallest). Another possibility would be to look for a sign change of the underlying function between neighboring grid points. We employ a faster method that is based on a quadtree subdivision algorithm as explained in Section 3.1.

We now present one way to implement the predictor step. *One-step methods* only employ one point (the point of the previous iteration) to predict a new point. The standard Euler method is a good example of a one-step method. However, one can envision the use of Hermites method which employs more than one point and is therefore called *multi-step method*. Although multi-step methods are usually more expensive, they lead to better, higher order approximations, reducing the number of necessary corrector steps.

Euler's method starts at a point which is known to be on the curve. Because this is a one-step method, each point on the curve will lead to exactly one new point on the curve. Since we execute many predictor-corrector iterations to obtain a point set that approximates our curve, it is just obvious to use the resulting point of the previous iteration as the initial point for the current iteration. We call this initial point $p_i = (x_i, y_i)$. We also want to have some control over the distance between the new point and

the initial point. That is, we want p_p to be located at a certain distance Δs from p_i and not at some unpredictable "end" of the curve. Δs is called the arc length parameter and it represents a parameterization of our curve. The linear approximation of the predicted point results in

$$\begin{aligned} x_p &= x_i + dx\Delta s \\ y_p &= y_i + dy\Delta s \end{aligned}, \tag{2}$$

where $p_p = (x_p, y_p)$ now depends on the arc length and the initial point. Since Δs is the euclidean distance between p_p and p_i, we need to make sure that:

$$dx^2 + dy^2 = 1 \tag{3}$$

Equation (3) defines a unit circle around the origin. Combining it with Equation (2) we are describing a circle of radius Δs and origin p_i. Next, we have to find the intersection points of that circle with our curve. We choose one of these points as our new predicted value. Since the origin of this circle lies already on the curve, we are guaranteed that our circle will intersect our implicit curve, unless the whole curve is contained within the circle. In that particular case, the choice of the distance Δs between neighboring points is not appropriate and needs to be changed. If Δs is in the order of the size of a pixel and the curve is still contained within such a small circle, the discretization of this curve would just be one pixel and we are done.

Assuming a closed curve, we will actually get an even number of intersection points. Choosing our step size (arc length) small enough, we will get exactly two intersection points (see Figure 1). However, in the case of a self intersecting curve or when a curve comes "close" to itself, we will get more than two intersection points, even if Δs is very small. We will explore these special cases in Section 2.2.

The second condition for the new predicted point is, of course, that it should lie on the curve defined by the underlying implicit function *f*. That means we require

$$f(x_p, y_p) = 0\,. \tag{4}$$

Since we are processing an arbitrary functions *f* we are not able to use Equation (4) directly to compute p_p. Instead, we substitute Equation (2) into Equation (4) and through a Taylor series expansion we are

able to compute an approximation to the actual point on the curve. This approximation serves as the predicted value p_p.

Let's have a look at the equations. The substitution results in

$$f(x_i + dx\Delta s, y_i + dy\Delta s) = 0. \tag{5}$$

Now the actual approximation takes place. We expand Equation (5) in a Taylor series and keep only the linear terms. That results in:

$$f(x_i, y_i) + \frac{\partial}{\partial x} f(x_i, y_i) dx\Delta s + \frac{\partial}{\partial y} f(x_i, y_i) dy\Delta s = 0 \tag{6}$$

The initial point was assumed to be on the curve, i.e. $f(x_i, y_i) = 0$. We conclude

$$\frac{\partial}{\partial x} f(x_i, y_i) dx\Delta s + \frac{\partial}{\partial y} f(x_i, y_i) dy\Delta s = 0 \tag{7}$$

Using Equation (3) and Equation (7), we can actually solve for the unknowns (dx, dy). Using a short-cut notation for the partial derivatives of $f(x_i, y_i)$, where $\frac{\partial}{\partial x} f(x_i, y_i)$ is replaced by f_x (and similarly for the partial derivative in y) the result can be written as

$$\begin{aligned} dx &= \delta \frac{f_y}{\sqrt{f_x^2 + f_y^2}} \\ dy &= -\delta \frac{f_x}{\sqrt{f_x^2 + f_y^2}} \end{aligned} \tag{8}$$

which is also proven by Baker and Overman [3]. Here δ stands for the sign plus or minus which represents the direction in which we continue to track the curve. In the case of the seed point we want to make sure, that we trace both directions. However, in the general case we don't want to step back into the direction we were coming from. A simple sign check of the scalar product of the direction, where we were coming from and our new direction, helps us determining the appropriate δ.

Although our new predicted value lies on a circle around the initial point, we only find a point near the curve because of the approximation in Equation (6). If we take a closer look at Equation (7), the one we actually used as an approximation to Equation (5), we realize that (dx, dy) describes the components

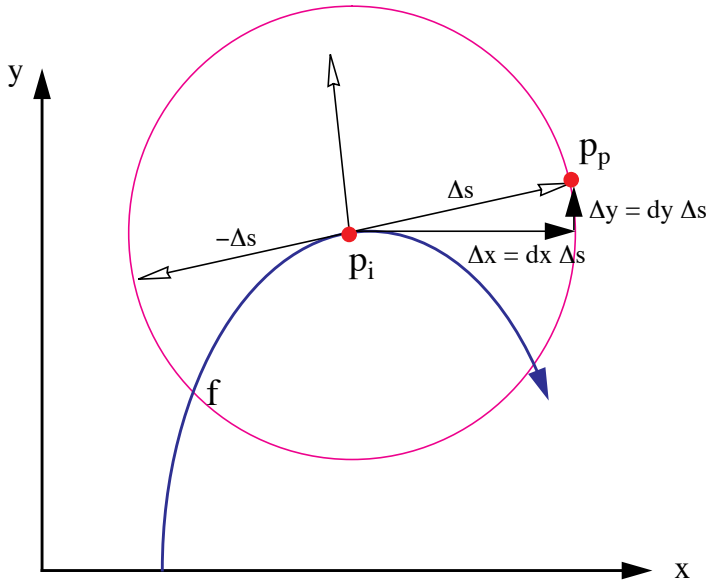

FIGURE 1. The predictor step. Here Δs is the step size we want to step along the curve starting at the initial point p_i. Because Equation (7) is only an approximation, the predicted value will not necessarily lie on the curve described by f.

of the tangent to the curve at the initial point. That leads us to a better geometric understanding of this process, captured in Figure 1.

2.2 Bifurcation Points

The intersection of the circle equation (originating in Equation (3)) with the implicit curve results usually in two points. As pointed out earlier, in case of a selfintersecting curve (Figure 2a) or if it just comes "close" to itself (Figure 2b) we expect at least four intersection points. These special points are also called *bifurcation points*. Looking at our formulas in Equation (7) and Equation (3), we are only able to compute two points, since it is an intersection of a quadratic (Equation (3)) curve with a linear curve (Equation (7)). In a case of a bifurcation point the linear approximation of the Taylor series as in Equation (7) will not be enough. In fact, from the implicit function theorem we can conclude, that the first partial derivatives will be zero for these bifurcation points. Therefore we cannot compute (dx, dy) as derived above. We have to include the second order terms of the Taylor series expansion in Equation (6) to get the appropriate results.

The more accurate approximation to Equation (5), replacing Equation (6), is now

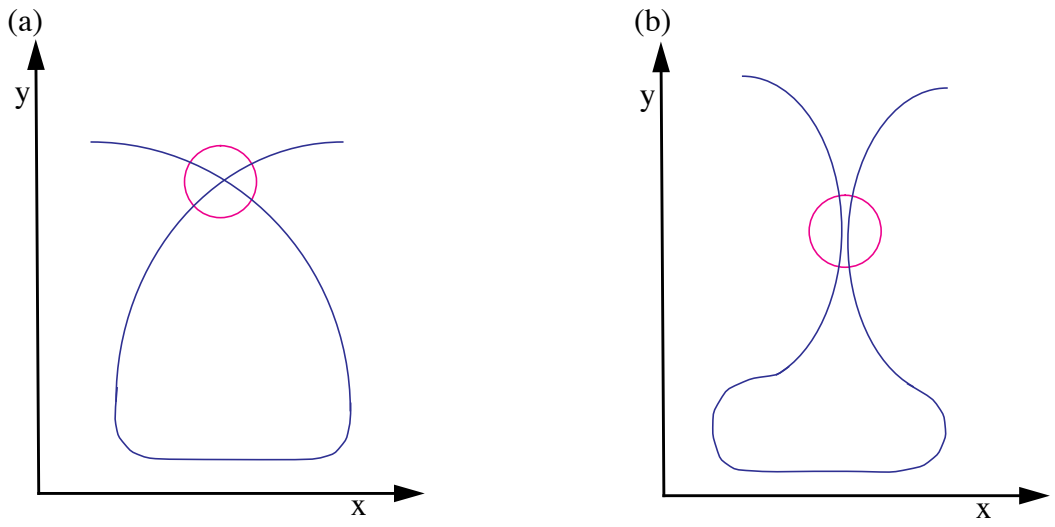

FIGURE 2. Possible cases of bifurcation points where the circle represents the actual step size. (a) The curve intersects itself; (b) The curve comes very “close” to itself.

$$\begin{aligned} &f(x_i, y_i) + f_x(x_i, y_i)dx\Delta s + f_y(x_i, y_i)dy\Delta s + \\ &f_{xx}(x_i, y_i)dx^2\Delta s^2 + 2f_{xy}(x_i, y_i)dxdy\Delta s^2 + f_{yy}(x_i, y_i)dy^2\Delta s^2 = 0 \end{aligned} \tag{9}$$

As already mentioned, the first three terms in the sum vanish, which leads to

$$f_{xx}(x_i, y_i)dx^2\Delta s^2 + 2f_{xy}(x_i, y_i)dxdy\Delta s^2 + f_{yy}(x_i, y_i)dy^2\Delta s^2 = 0 \tag{10}$$

In order to increase the readability of the following formulas, we use the following notation:

$$\begin{aligned} A &= f_{xx}(x_i, y_i) \\ B &= f_{xy}(x_i, y_i) \\ C &= f_{yy}(x_i, y_i) \end{aligned} \tag{11}$$

If we consider Equation (10) as a quadratic equation in the quotient q defined by dx/dy, we get the following solution:

$$q_{1,2} = \frac{dx}{dy} = \frac{-B \pm \sqrt{B^2 - AC}}{A} = \frac{C}{-B \mp \sqrt{B^2 - AC}} \tag{12}$$

To actually compute dx and dy, we substitute this result in Equation (3), which results in

$$\begin{pmatrix} dx \\ dy \end{pmatrix} = \pm \begin{pmatrix} \sqrt{\dfrac{1}{\dfrac{1}{q_{1,2}^2}+1}} \\ \sqrt{\dfrac{1}{q_{1,2}^2+1}} \end{pmatrix} = \pm \begin{pmatrix} \sqrt{\dfrac{C^2 - AC + 2B\,(B \pm \sqrt{B^2 - AC})}{(A-C)^2 + 4B^2}} \\ \sqrt{\dfrac{A^2 - AC + 2B\,(B \mp \sqrt{B^2 - AC})}{(A-C)^2 + 4B^2}} \end{pmatrix} \tag{13}$$

where any arbitrary combination of plus and minus from Equation (12) and Equation (13) results in four different distinct direction vectors (dx, dy). We will store all four directions in a list and follow their branches one at a time. This is explained in greater detail in Section 3.

We discussed the usual case of two intersection points and studied in detail what happens when these formulas fail when we encounter bifurcation points. Unfortunately, it could happen that we do not have four intersection points, but six, eight, or more. That means, that actually more then the two branches, as shown in Figure 1, meet at the same area. In that case, we realize that the second partial derivatives in Equation (9) vanishes also, leaving us with no choice other than including even higher order terms in the approximation of the Taylor series. This implies that we will have to solve higher order polynomials to produce the expected number of solutions.

2.3 The Corrector Step

Although we have tried hard to predict a new point p_p on the curve, it is not guaranteed to fall accurately on the curve. We can only hope to find a point that is "very close" to the curve. The goal of the *corrector step* is to find a "closest" point on the curve, starting at p_p. Among the many methods for finding zero points of functions, the Newton method is very stable and fast, and therefore very common. The Newton method is a numerical iteration method which converges to a zero point p^* of a function f, starting at a point p_0 that should be fairly close to the zero point already. In our case, the starting point for the Newton method will be p_p – the result of the predictor step.

The predictor step produced an initial value $p_0 = p_p = (x_p, y_p) = (x_0, y_0)$ for the iteration method. At each iteration step we take the new and improved value from the previous iteration and try to improve it even further, so that we actually end up on the curve itself. The result of the current iteration can be written as:

$$p_{k+1} = \begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k + \Delta x \\ y_k + \Delta y \end{pmatrix}, \tag{14}$$

where k marks the iteration number.

But how can we determine $(\Delta x, \Delta y)$? Here we exploit the same idea as we did in Euler's Method. Of course, we want that our new point (x_{k+1}, y_{k+1}) lies on the curve. So we add the condition that p_{k+1} is a curve point:

$$f(x_{k+1}, y_{k+1}) = 0. \tag{15}$$

The same ideas and manipulations as used for Equation (4) will lead us towards a solution. Substituting Equation (14) into Equation (15) and using a Taylor series expansion of f will result in:

$$f_x(x_k, y_k)\Delta x + f_y(x_k, y_k)\Delta y = -f(x_k, y_k). \tag{16}$$

That is, in fact, the formula for Newton's method for a two dimensional function. Since this is one equation with two unknowns, we have the freedom and duty to specify another condition, we want our iterate to satisfy. First of all, we want to make sure that our iteration process converges. Other than that, it would be nice if we could guarantee some sort of constant discretization of our curve. That is, we do not want the distance between samples to vary too much. Both of these ideas can be realized if we restrict our convergence to move in a right angle away from our previous iterate. To be more specific, the triangle formed by the points p_i, p_k and p_{k+1} should form a right angled triangle. Here, p_i is the initial point on the curve – the one we used in the predictor step to compute the point p_p. p_k and p_{k+1} are the previous and current iterates. In Figure 1 it was shown that this condition on our new iterate results in a somewhat circular movement around the initial point p_i. Because of this circular movement, we stay in a certain proximity to p_i, which increases the chances of convergence.

More formally, we want to add the condition that the vector $\overrightarrow{p_k p_i}$ is perpendicular to $\overrightarrow{p_k p_{k+1}}$. That is expressed in the following manner:

$$(p_i - p_k) \bullet (p_{k+1} - p_k) = \begin{bmatrix} x_i - x_k \\ y_i - y_k \end{bmatrix}^T \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \Delta x\,(x_i - x_k) + \Delta y\,(y_i - y_k) = 0 \tag{17}$$

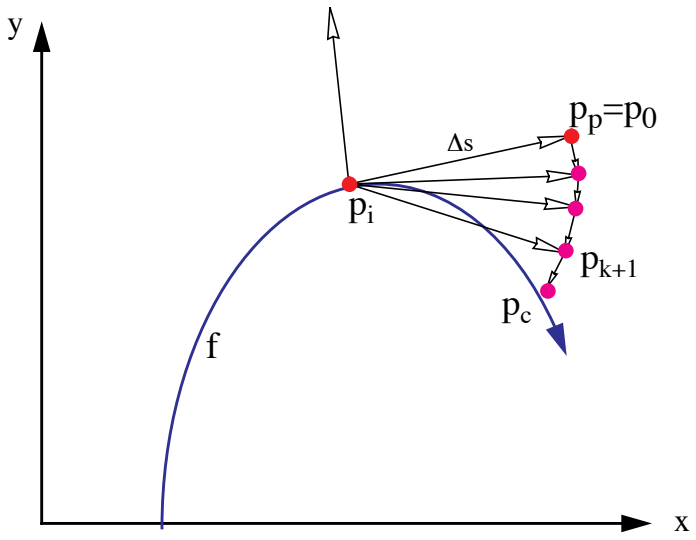

FIGURE 3. The corrector step. The Newton iterations and a forced circular movement of the iterates p_{k+1} converge towards an actual point on the curve p_c.

Equation (16) together with Equation (17) uniquely describe a new, corrected iterate p_{k+1}, which concludes the Corrector step. A geometric interpretation of these steps is illustrated in Figure 3.

3. The Algorithm

The pseudo code for our numerical algorithm is given in Figure 4. The algorithm starts by looking for a seed point (using a space subdivision method, as described in Section 3.1). The initial point will be most likely on some arbitrary part of the curve. Therefore, we will usually have two possible directions we can follow. We record both in a list for later retrieval (Line 3). At a bifurcation point a pixel will have four branches leaving it. In this case we will also record all possible branch directions in this list (Line 18). After retrieving a starting point for a new branch (Line 5) we trace the branch until we find another bifurcation point or until we reach the limits of our display device. In the case of a closed curve we also need to check when we get back to an already traced branch of the curve. A simple way to do this is explained in Section 3.5.

```
1. find an initial point p_i on the surface;
2. determine initial step size Δs;
3. record (p, Δs, bifurlist);                    /* for both initial direction */
4. repeat
5.     get p_i, Δs from bifurlist
6.     repeat                                    /* continue current branch */
7.         repeat                                /* find a reliable new point on the curve */
8.             p_p := Predictor(p_i)
9.             p_c := Corrector(p_p)
10.            C := approx_curvature(p_c, p_p)
11.            Δs := Δs/2
12.        until (p_c converges AND 1/(2C) < Δs) OR bifur_test(p_c)
13.        Δs := Δs*2;
14.        if 1/(2C) >Δs then
15.            increase Δs;
16.        draw_line(p_c, p_i);
17.        if bifur_test(p_c) then
18.            record(p_c, Δs, bifurlist)
19.        p_i := p_c;
20.    until (p_c is in the covered region or outside the screen OR bifur_test(p_c));
21. until bifurlist is empty
```

FIGURE 4. pseudo code for the curve tracking algorithm.

For each new predictor point we need to make sure that the corrector didn't fail (Line 7 through Line 12). Once we have gained confidence about our corrected point, we check whether we can adjust the step size (Line 15). We connect the previous point to the current point by a simple line (Line 16). If we found a new bifurcation point, we need to record it (Line 18) and quit this loop. In any other case we repeat this procedure and find another point on the same branch.

We now describe in more detail the implementation of some of the more involved segments of the algorithm.

3.1 Finding a Starting Point

If a seed point is not provided by the user, we employ a space subdivision approach. We divide the image region into four quadrants and compute the function at each corner of the quadrants. If the sign of the function is the same at all corners, we will apply the same subdivision to all four quadrants in a breath-first scheme. If we do find a sign change, we continue to subdivide this specific quadrant in a depth-first scheme up to the pixel- or even subpixel level, depending on how accurate our initial point should be.

3.2 Implementing the Corrector Step

Since the predicted point is only an approximation of a curve point, the corrector step is employed to converge to an actual point on the curve. Because of the approximations in our formulas, the Newton corrector step (explained in Section 2.3) usually needs to be repeated a few times. That is done within the Corrector procedure in Line 9. But how many times should we repeat the Newton step? Of course, we have to fight round off errors caused by the discrete computer arithmetic as well. The quality of our discretized curve greatly depends on our stopping criteria, as well as the successful completion of our algorithm. Therefore, we need to know when $f(p_{k+1})$ gets close enough to zero. The problem is that *close* means very different things for different functions. If we had more knowledge about the function, we could employ an absolute error test, like $f(p_{k+1}) < \varepsilon$. For a general algorithm, we will have no idea how small ε needs to be to assure spatial proximity. This notion of *being close* to the curve might also change at different ranges for the curve.

The method we chose adapts the value of ε and utilizes the fact that we draw to a finite grid. Every once in a while (e.g. every tenth time we call the Corrector procedure) we check the quality of the corrected point p_c. For this quality check we define a rectangular region centered at p_c and check for a sign change of the function f at the corner points. The size of this rectangular region can be a parameter L specified by the user or simply the current grid size. If there is a sign change in the corner points, we know that our curve will be at most $L/\sqrt{2}$ away. Therefore, we could relax our stopping criteria and set ε to $1.1|f(p_{k+1})|$. On the other hand, if we find that our corrected value isn't close enough to our curve, we should strengthen our stopping condition and set ε to $10^{-2}|f(p_{k+1})|$ and redo the corrector

step. This finally results in a relative (adaptive) error test, which is certainly superior than an absolute (static) error test.

3.3 Determining The Step Size Δs

The other crucial and very sensitive parameter of this algorithm is the step size Δs . One big advantage of our algorithm is the freedom to choose the step size. When the curve shows a small curvature we will choose a fairly large Δs , allowing the algorithm to step quickly through this “almost linear” part of the curve. On the other hand, when the curve changes rather quickly, i.e. the curvature of the function is high, we have to decrease Δs appropriately. This adaptive mechanism gives us the ability to capture the nature of our curve in greater detail when necessary.

The mathematical definition of the curvature is

$$C(p) = \lim_{q \to p} \frac{\alpha(p) - \alpha(q)}{\widehat{pq}}, \tag{18}$$

where $\alpha(p)$ denotes the angel of the tangent with the x-axis and $\widehat{pq}$ denotes the arc length between the two points. For implicit functions, C can be computed as

$$C = \frac{-f_y^2 f_{xx} + 2 f_x f_y f_{xy} - f_x^2 f_{yy}}{\left(f_x^2 + f_y^2\right)^{3/2}} \tag{19}$$

(See [6]). We compute first derivatives anyway for the Newton corrector, but would like to avoid computing second derivatives. Therefore, we use an approximation to the curvature that is based on Equation (18), where we use the current and previous point on the curve. Since we have the first derivatives already computed we can precisely compute the tangent angles. The arc length between the two points is simply approximated with the current step size. To compute the initial step size we could either use Equation (19) or just use the grid size. Choosing the initial step size for a bifurcation point can be a problem. If we are stepping too far, we might ‘loose track’ of the current branch. If we are stepping too close, the corrector might converge to a different, neighboring branch. Therefore we left it up to the user to choose this initial step size.

Another indicator of the quality of our step size would be the number of corrector steps we had to take before we converged to a curve point. In our predictor step we are basically following the tangent of the curve. If the curve is fairly linear the predicted point p_p will be almost on the curve, resulting in just very few corrector steps. If the curve changes its direction rather fast, p_p will not be the best guess and we expect more correction steps. Therefore, if we use at most one Newton step to find a point on the curve, we simply double the step size.

3.4 Detecting Bifurcation Points

Bifurcation points are points where the curve intersects itself. At these points the first derivatives of the function evaluate to zero. Of course, it will be very unlikely to find the mathematically exact bifurcation point (in case one exists), because of the numerical nature of the algorithm. Therefore, we need to keep watching the first derivatives of a function for regions where they are *close* to zero. We have seen already how difficult such a test can be (see Section 3.2). Therefore, whenever we find a sign change in one of the partial derivatives (from the previous to the current point) we have good reason to suspect that there might be a bifurcation point close by. As a result, we decrease our step size and repeat the last predictor/corrector step until the step size is smaller than a given size, which can be defined by the user. If there is still reason to believe that there might be a bifurcation point, we create a rectangular region around the current point on the curve. Similar to the test we performed in Section 3.2, we test whether there is a sign change in the partial derivatives f_x and f_y. If we find a sign change in both partial derivatives we conclude that we detected a bifurcation point. Otherwise, we conclude that we probably found a turning point of the curve.

3.5 Stopping Criteria

We assume that we deal with a closed curve. Therefore, we need to find out if and when we returned to a part of a branch we have traced already. If we do not include this exit test (in Line 20), we will trace the curve over and over again, ending up in an infinite loop.

Because of the nature of a numeric algorithm it is very unlikely to return to the exact same point from where we started. Therefore, we cannot just test for equality. We also do not want to test the proximity

of the new point p_c to every one of the computed values along the curve. That might be possible for the first few values, when the cost is still low, but certainly not for arbitrary curves, where we have to compute many points to reasonably approximate the curves shape. The other problem is again the meaning of "very close". To solve this problem, we maintain a counter that counts the number of times we attempt to draw a pixel that has been drawn already. When the algorithm computes a point on the curve that is mapped to a pixel that was not drawn before, we set the counter to zero. Every time we compute a new point on the curve within a pixel that was previously drawn, we increase a counter by one. If the counter's value exceeds some threshold, the algorithm stops pursuing this path.

TABLE 1. Performance results comparing our algorithm. The empty cells denote cases where Chandler's algorithm could not draw the curve.

Figure	# Function evaluation per pixel		# pixels per predictor	# pixels per predictor
	Our Method	Chandler		
5a	1.89	2.68	6.21	2.55
5b	2.01	2.80	6.79	3.20
5c	3.27	2.92	3.51	2.49
6a	2.88	–	6.66	5.04
6b	4.18	–	4.05	3.67
6c	4.55	–	2.38	2.30

4. Results

We applied our algorithm to many different functions with and without bifurcation points and found that they were drawn correctly in any image resolution of 128^2 to 512^2. For images of low resolution (32^2 and sometimes 64^2) the user may have to explicitly specify a smaller step size. We have chosen three very common functions without a bifurcation point (Figure 5) and three interesting functions with many bifurcation points (Figure 6). Some interesting statistics for each group of functions is summarized in Table 1 and compared to Chandlers algorithm [7]. The test cases where drawn onto a 5122 image. Drawing times, for all examples, was under 0.2 seconds on a Silicon Graphics Crimson (150Mhz R4400). In the table we also provide the average number of pixels per one predictor step (showing the adaptive step size), and the average number of correct steps per predictor.

In general, the number of function evaluations per pixel for low resolutions is expectably high, since the curvature of a curve considered within a pixel is larger than if the same curve is subdivided into smaller parts (i.e. for smaller pixel sizes). Therefore, the step size (in terms of the number of pixels) is smaller and the algorithm must perform more calculations on a per pixel basis. For high resolutions we have much fewer function evaluations per pixel than Chandler's algorithm without loss of quality. This demonstrates the power of the adaptive step size.

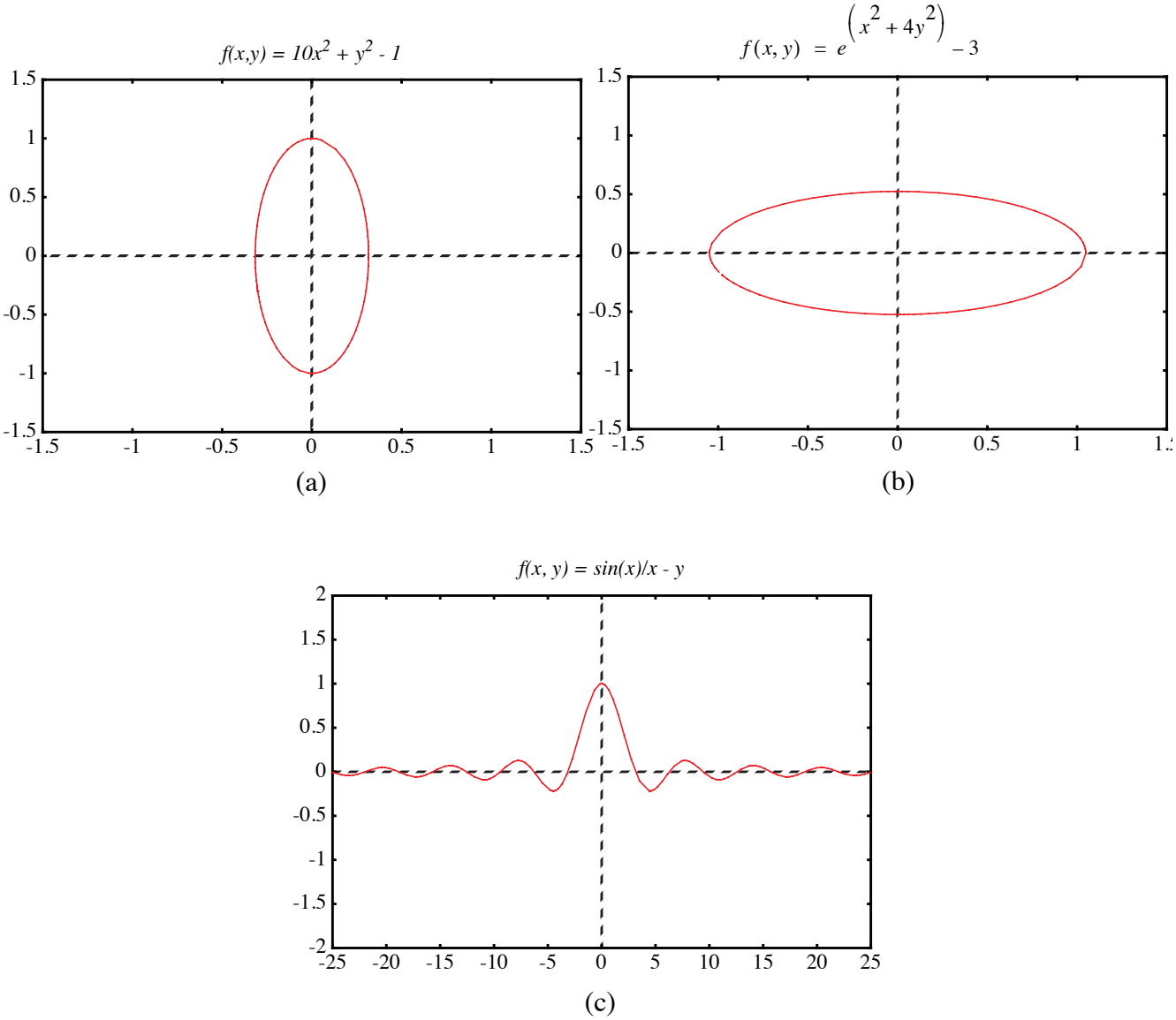

FIGURE 5. Curves without bifurcation points.

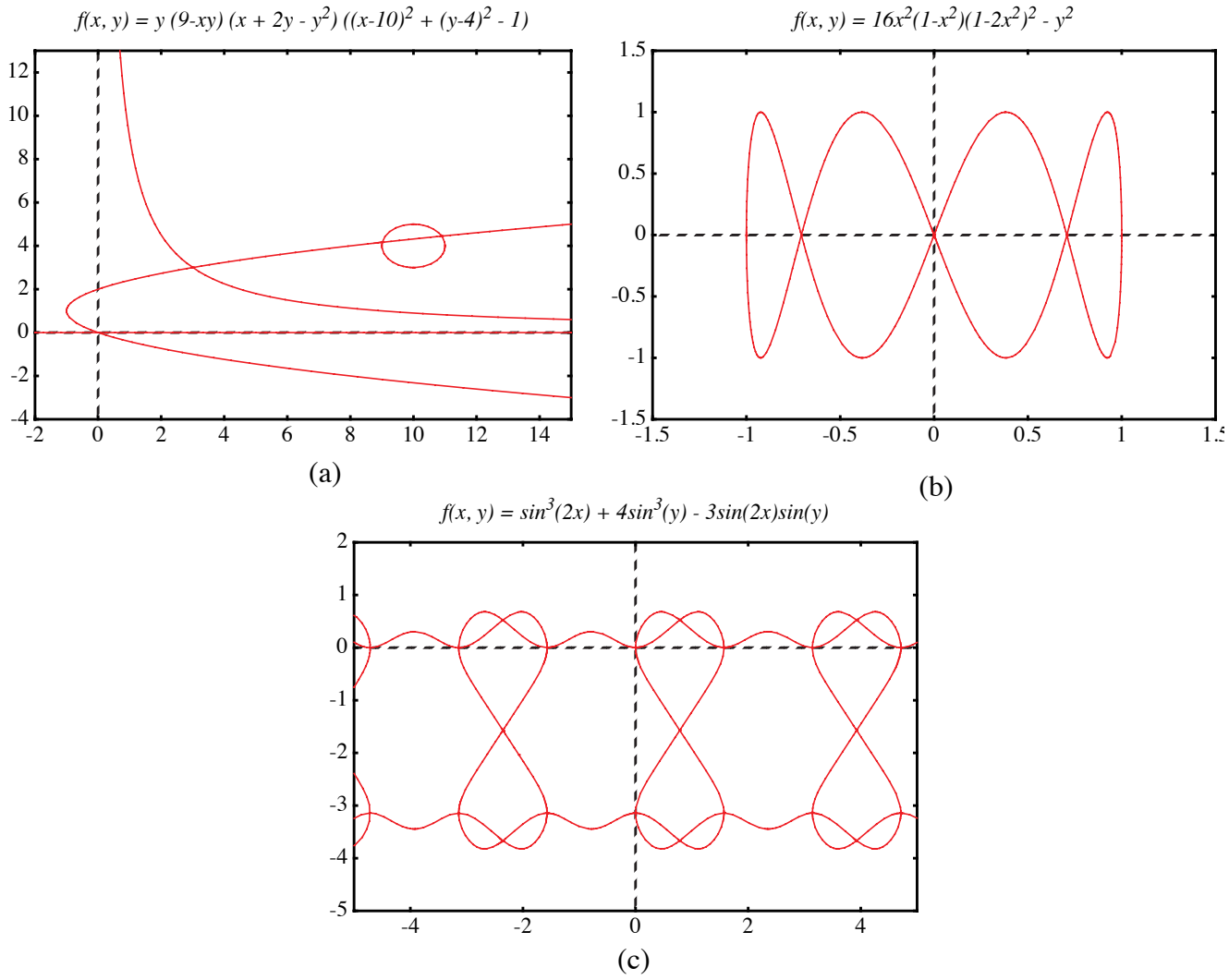

FIGURE 6. Curves with bifurcation points.

5. Conclusions

We presented here a numerical algorithm that is able to draw a large family of implicit functions. Through an adaptive step size design our performance is greatly optimized. We have shown that our approach outperforms previous tracking algorithms.

Our future research includes the improvement of the accuracy of the predictor step. Since we already know the derivatives at the previous point and, most of the time, also have this information available for the point before that, we could use the Hermite interpolation scheme and approximate our curve with a

polynom of third degree. This higher order approximation of our curve should result in fewer corrector steps. Since the higher order polynomial is computed only at the predictor step, saving several corrector steps, we expect a more efficient algorithm. A similar improvement can be made when connecting a new point on the curve with the previous one. Instead of a line through these point, one could use a higher order polynomial. Improvements to the accuracy of the corrector procedure are also possible, for example, by using a better corrector than Newton's. Finally, we plan to explore the extension of our methods to 3D voxelization of implicit surfaces.

6. References

[1] Allgower E. and S. Gnutzmann, "Simplicial Pivoting for Mesh Generation of Implicitly Defined Surfaces", Computer Aided Geometric Design, 8(4), 1991, pp. 30.

[2] Arvo J. and K. Novins, "Iso-Contour Volume Rendering", Proceedings of 1994 Symposium On Volume Visualization.

[3] Baker G. and E. Overman, "The Art of Scientific Computing", Draft Edition, The Ohio State Bookstore.

[4] Blinn J.F., "A Generalization of Algebraic Surface Drawing", ACM Transactions on Graphics, 1(3), 1982, pp. 235-256.

[5] Bloomenthal J., "Polygonization of Implicit Surfaces", Computer Aided Geometric Design, 5(4), 1988, pp. 341-356.

[6] Bronstein I. N. and Semendjajew, K. A. "Taschenbuch der Mathematik", BSB B.G.Teubner Verlagsgesellschaft, Leipzig, 1985.

[7] Chandler R.E., "A Tracking Algorithm for Implicitly Defined Curves", IEEE Computer Graphics and Applications, 8(2), March 1988, pp. 83-89.

[8] Foley J. D., A. van Dam, S. K. Feiner and J. F. Hughes, "Computer Graphics Principles and Practice", Addison-Wesley, 1990.

[9] Jordan B.W., W.J. Lennon, and B.D. Holm, "An Improved Algorithm for the Generation of Nonparametric Curves", IEEE Trans. Computers, Vol. C-22, 1973, pp. 1052-1060.

[10] Hall M. and J.Warren, "Adaptive Polygonization of Implicitly Defined Surfaces", IEEE Computer Graphics and Applications, 10(6), November 1990, pp. 33-42.

[11] Hanrahan P., "Ray Tracing Algebraic Surfaces", Computer Graphics (SIGGRAPH'83 Proceedings), 17(3), 1983, pp. 83-90.

[12] J.C. Hart. Ray Tracing Implicit Surfaces. WSU Technical Report EECS-93-014. Chapter in "Design, Visualization, and Animation of Implicit Surfaces," ACM SIGGRAPH '93 intermediate course notes, J. Bloomenthal and B. Wyvill, eds., Aug. 1993.

[13] Hobby J.D., "Rasterization of Nonparametric Curves", ACM Transactions on Graphics, 9(3), July 1990, pp. 262-277.

[14] Kalra D. and A.H. Barr, "Guaranteed Ray Intersections with Implicit Surfaces", Computer Graphics (SIGGRAPH'89 Proceedings), 23, 1989, pp. 297-306.

[15] Lorensen W. and H. Cline, “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”, Computer Graphics, 21(4), 1987, pp. 163-169.

[16] Sederberg T.W., D.C.Anderson, and R.N.Goldman, “Implicitization, Inversion, and Intersection of Planar Rational Cubic Curves”, Computer Vision, Graphics, and Image Processing, 31(1), 1985, pp. 89-102.

[17] Sederberg T.W. and A.K.Zundel, “Scan Line Display of Algebraic Surfaces”, Computer Graphics (SIGGRAPH’89 Proceedings), 23(4), 1989, pp. 147-156.

Efficient Rasterization of Implicit Functions

Torsten Möller and Roni Yagel

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio

{moeller, yagel}@cis.ohio-state.edu

Abstract

Implicit curves are widely used in computer graphics because of their powerful features for modeling and their ability for general function description. The most popular rasterization techniques for implicit curves are space subdivision and curve tracking. In this paper we are introducing an efficient curve tracking algorithm that is also more robust then existing methods. We employ the Predictor-Corrector Method on the implicit function to get a very accurate curve approximation in a short time. Speedup is achieved by adapting the step size to the curvature. In addition, we provide mechanisms to detect and properly handle bifurcation points, where the curve intersects itself. Finally, the algorithm allows the user to trade-off accuracy for speed and vice a versa. We conclude by providing examples that demonstrate the capabilities of our algorithm.

1. Introduction

Implicit surfaces are surfaces that are defined by implicit functions. An implicit function f is a mapping from an n dimensional space R^n to the space R, described by an expression in which all variables appear on one side of the equation. This is a very general way of describing a mapping. An n dimensional implicit surface S is defined by all the points p in R^n such that the implicit mapping f projects p to 0. More formally, one can describe S by

$$S_f = \{p \mid p \in R^n, f(p) = 0\} . \quad (1)$$

Often times, one is interested in visualizing *isosurfaces* (or *isocontours* for 2D functions) which are functions of the form $f(p) = c$ for some constant c.

Different classes of implicit surfaces are of importance. Implicit surfaces that are defined by a polynomial are called *algebraic surfaces*. Since almost all continuous functions can be approximated by polynomials, algebraic surfaces represent a powerful class. Since polynomials have been studied and understood fairly well, one has a better mathematical handle at algebraic surfaces.

Another special class of implicit surfaces have density functions as their defining functions. Density functions decrease exponentially with increasing distance to a certain center point. The models where these functions find usage usually arise in the study of physical phenomena, e.g. atom models. Blinn [4] was one of the first to try to visualize these surfaces on a computer. They became to be known as Blinn's "*Blobby Model*" or *Metaballs*.

Yet another model which is gaining importance is the *discrete data set*. Such data set is commonly generated by various medical scanners such as MRI and CT. These data sets can also be seen as implicit functions, where $f(p) = v$ and v is the scanned value at the location p. We assume that if p is not on a grid point then some sort of interpolation from neighboring grid points defines v at p.

Rendering implicit surfaces is very difficult and computationaly very expensive. Usually existing algorithms to render implicit surfaces work only for special cases or special sets of functions. In fact, to date there is no such algorithm that can render arbitrary implicit surfaces accurately and in a stable and timely manner.

Algorithms for rendering 3D implicit surfaces can be classified into three groups: representation transforms, ray tracing, and surface tracking. The first approach tries to transform the problem from implicit surfaces to another representation which is easier to display such as polygon meshes [1][5][10][15] or parametric representations [16]. The second approach to rendering of implicit surfaces is to ray-trace them. Efficient ray-object intersection algorithms have been devised for algebraic surfaces [11], metaballs [4], and a class of functions for which Lipschitz constants can be efficiently computed [12][14]. The third approach to rendering implicit functions is to scan convert or discretize the function. Seder-

berg and Zundel [17] introduced such a scan line algorithm for a class of implicit functions that can be represented by Bernstein polynomials.

When it comes to drawing 2D implicit functions (curves) an attractive approach is to follow the path of the curve. The algorithm starts at a seed point on the curve. It then looks at the pixels adjacent to the seed point to see if any of them is the next pixel along the curve. Various algorithms exist which mainly differ in the method they use to find the next pixel along the curve.

One way to find the next pixel is based on *digital difference analysis* (DDA). A lot of work has been done in this field and the most popular algorithm applying this technique are Bresenham's and the midpoint line and circle drawing algorithms [8]. The rate of change in x and y is computed through fast integral updates of dx and dy, which are then used as decision variables to determine the next pixel center. This idea was generalized for algebraic curves by Hobby [13]. One problem of this approach is that the inherent integer arithmetic can only be exploited for a small set of algebraic functions. Furthermore, this algorithm is not able to deal properly with self intersecting curves. Hobby assumes that these intersection points are given by the user.

A different way to continue a curve would be to evaluate the underlying function at the center of the neighboring pixels to find the one with the smallest absolute function value. This idea has been implemented by Jordan et. al. [9] and was improved by Chandler [7]. Since there are only eight neighboring pixels, eight function evaluations are the maximum needed to find the next pixel on the curve. Since we will very likely be traveling in the same direction we have been traveling when we picked the current pixel, Chandler tests first the pixel continuing in the same direction, then its neighbors etc. For most 'nicely' behaving functions two to three function evaluations per pixel will be sufficient. To increase accuracy, Chandler prefers to look for a sign change rather than the smallest absolute value. Therefore, he evaluates the function not at pixel centers but rather halfway in between pixels. Although the algorithm does not depend on the actual description of the function and therefore is applicable to arbitrary functions, it is still not able to deal with self intersecting functions. To deal with such points the algorithm requires user intervention. Alternatively, it includes a brute-force Monte Carlo method to visualize the positive and negative regions of the function. Finally, if the curve enters and leaves the pixel in-

between the same two neighboring sample points, this algorithm will not be able to register a sign change and therefore fail.

For density functions, Arvo et. al. applied a numerical method known as the predictor-corrector method [2]. Their main goal is the rendering of iso-curves of medical data sets. In this paper we introduce an efficient and more robust predictor-corrector method. Our algorithm deals with self intersecting curves properly and increases the efficiency of the algorithm to less than one function evaluation per pixel on the average. Finally, unlike existing curve tracking methods, our algorithm can be applied to a large family of implicit functions including selfintersecting functions with at most two branches at the intersection point.

2. The Predictor-Corrector Method

The predictor-corrector method is a well known numerical algorithm which has its origin in solving ordinary differential equations and bifurcation theory. Arvo and Novins [2] used a two dimensional standard method for extracting isocurves in a 2D medical image. Although the predictor-corrector method offers ways to detect and properly deal with self intersecting curves and curves “very close” to themselves, Arvo and Novins did not explore these capabilities.

The predictor-corrector method is a numerical continuation method (tracking algorithm). The goal is to find a point set which represents a discretized approximation of a curve that is defined by some implicit function as in Equation (1). As the name suggests, the predictor-corrector method consists of two steps. In the predictor step we take a (more or less sophisticated) guess of a point on the curve, denoted by p_p. Depending on how much thought and work we put into predicting the point p_p, it will be closer or farther away from an actual point on the curve. Depending on the assumption we make about the curve and its actual shape, we will not be able to always accurately predict an exact point on the curve. In the corrector step we use our predicted point p_p as a starting point for an iterative method that converges to an actual point on the curve. We denote this new, corrected point, by p_c. We repeat these two steps until we tracked the whole curve. The resulting discrete point set can then be displayed on a screen.

There are many different ways to implement the predictor or corrector steps. We use the standard Euler-Newton method with an adaptive step size and introduce techniques to handle selfintersecting curves.

2.1 The Predictor Step

The focus of the predictor step is to find a point p_p that is very close or even on the curve, so that the costs for the corrector step are reduced or even eliminated. If we already know points on the curve, we can approximate the curve through a polynomial and get a new approximate for this curve through extrapolation or interpolation.

Finding the very first point on the curve is not always easy. One usually has some understanding of the kind of function one tries to discretize and can therefore provide an initial seed value to the algorithm. If this is not possible, one needs to apply a brute force algorithm or some other heuristic to find an initial seed value. The image space is limited (usually a screen in computer graphics). Therefore one rather brute-force method would be to impose a grid onto the screen and evaluate the underlying implicit function at each grid point to choose a closest point (where $|f(p)|$ becomes smallest). Another possibility would be to look for a sign change of the underlying function between neighboring grid points. We employ a faster method that is based on a quadtree subdivision algorithm as explained in Section 3.1.

We now present one way to implement the predictor step. *One-step methods* only employ one point (the point of the previous iteration) to predict a new point. The standard Euler method is a good example of a one-step method. However, one can envision the use of Hermites method which employs more than one point and is therefore called *multi-step method*. Although multi-step methods are usually more expensive, they lead to better, higher order approximations, reducing the number of necessary corrector steps.

Euler's method starts at a point which is known to be on the curve. Because this is a one-step method, each point on the curve will lead to exactly one new point on the curve. Since we execute many predictor-corrector iterations to obtain a point set that approximates our curve, it is just obvious to use the resulting point of the previous iteration as the initial point for the current iteration. We call this initial point $p_i = (x_i, y_i)$. We also want to have some control over the distance between the new point and

the initial point. That is, we want p_p to be located at a certain distance Δs from p_i and not at some unpredictable "end" of the curve. Δs is called the arc length parameter and it represents a parameterization of our curve. The linear approximation of the predicted point results in

$$\begin{aligned} x_p &= x_i + dx\Delta s \\ y_p &= y_i + dy\Delta s \end{aligned}, \tag{2}$$

where $p_p = (x_p, y_p)$ now depends on the arc length and the initial point. Since Δs is the euclidean distance between p_p and p_i, we need to make sure that:

$$dx^2 + dy^2 = 1 \tag{3}$$

Equation (3) defines a unit circle around the origin. Combining it with Equation (2) we are describing a circle of radius Δs and origin p_i. Next, we have to find the intersection points of that circle with our curve. We choose one of these points as our new predicted value. Since the origin of this circle lies already on the curve, we are guaranteed that our circle will intersect our implicit curve, unless the whole curve is contained within the circle. In that particular case, the choice of the distance Δs between neighboring points is not appropriate and needs to be changed. If Δs is in the order of the size of a pixel and the curve is still contained within such a small circle, the discretization of this curve would just be one pixel and we are done.

Assuming a closed curve, we will actually get an even number of intersection points. Choosing our step size (arc length) small enough, we will get exactly two intersection points (see Figure 1). However, in the case of a self intersecting curve or when a curve comes "close" to itself, we will get more than two intersection points, even if Δs is very small. We will explore these special cases in Section 2.2.

The second condition for the new predicted point is, of course, that it should lie on the curve defined by the underlying implicit function *f*. That means we require

$$f(x_p, y_p) = 0\,. \tag{4}$$

Since we are processing an arbitrary functions *f* we are not able to use Equation (4) directly to compute p_p. Instead, we substitute Equation (2) into Equation (4) and through a Taylor series expansion we are

able to compute an approximation to the actual point on the curve. This approximation serves as the predicted value p_p.

Let's have a look at the equations. The substitution results in

$$f(x_i + dx\Delta s, y_i + dy\Delta s) = 0. \tag{5}$$

Now the actual approximation takes place. We expand Equation (5) in a Taylor series and keep only the linear terms. That results in:

$$f(x_i, y_i) + \frac{\partial}{\partial x} f(x_i, y_i) dx\Delta s + \frac{\partial}{\partial y} f(x_i, y_i) dy\Delta s = 0 \tag{6}$$

The initial point was assumed to be on the curve, i.e. $f(x_i, y_i) = 0$. We conclude

$$\frac{\partial}{\partial x} f(x_i, y_i) dx\Delta s + \frac{\partial}{\partial y} f(x_i, y_i) dy\Delta s = 0 \tag{7}$$

Using Equation (3) and Equation (7), we can actually solve for the unknowns (dx, dy). Using a short-cut notation for the partial derivatives of $f(x_i, y_i)$, where $\frac{\partial}{\partial x} f(x_i, y_i)$ is replaced by f_x (and similarly for the partial derivative in y) the result can be written as

$$\begin{aligned} dx &= \delta \frac{f_y}{\sqrt{f_x^2 + f_y^2}} \\ dy &= -\delta \frac{f_x}{\sqrt{f_x^2 + f_y^2}} \end{aligned} \tag{8}$$

which is also proven by Baker and Overman [3]. Here δ stands for the sign plus or minus which represents the direction in which we continue to track the curve. In the case of the seed point we want to make sure, that we trace both directions. However, in the general case we don't want to step back into the direction we were coming from. A simple sign check of the scalar product of the direction, where we were coming from and our new direction, helps us determining the appropriate δ.

Although our new predicted value lies on a circle around the initial point, we only find a point near the curve because of the approximation in Equation (6). If we take a closer look at Equation (7), the one we actually used as an approximation to Equation (5), we realize that (dx, dy) describes the components

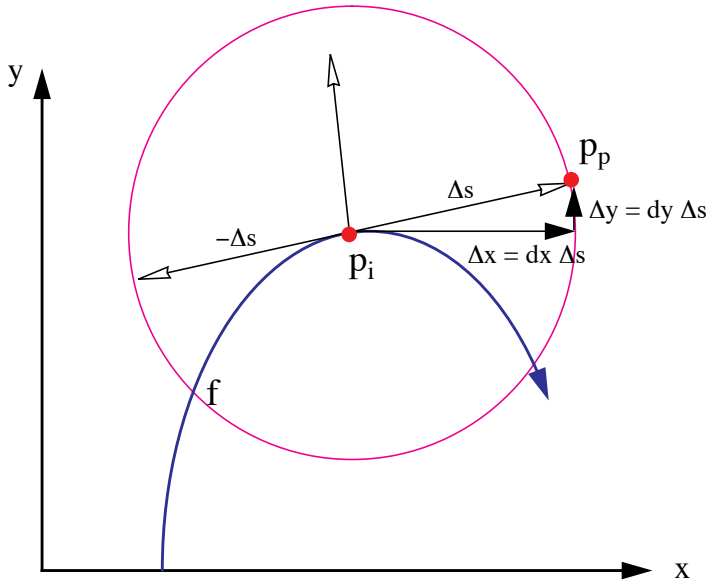

FIGURE 1. The predictor step. Here Δs is the step size we want to step along the curve starting at the initial point p_i. Because Equation (7) is only an approximation, the predicted value will not necessarily lie on the curve described by f.

of the tangent to the curve at the initial point. That leads us to a better geometric understanding of this process, captured in Figure 1.

2.2 Bifurcation Points

The intersection of the circle equation (originating in Equation (3)) with the implicit curve results usually in two points. As pointed out earlier, in case of a selfintersecting curve (Figure 2a) or if it just comes "close" to itself (Figure 2b) we expect at least four intersection points. These special points are also called *bifurcation points*. Looking at our formulas in Equation (7) and Equation (3), we are only able to compute two points, since it is an intersection of a quadratic (Equation (3)) curve with a linear curve (Equation (7)). In a case of a bifurcation point the linear approximation of the Taylor series as in Equation (7) will not be enough. In fact, from the implicit function theorem we can conclude, that the first partial derivatives will be zero for these bifurcation points. Therefore we cannot compute (dx, dy) as derived above. We have to include the second order terms of the Taylor series expansion in Equation (6) to get the appropriate results.

The more accurate approximation to Equation (5), replacing Equation (6), is now

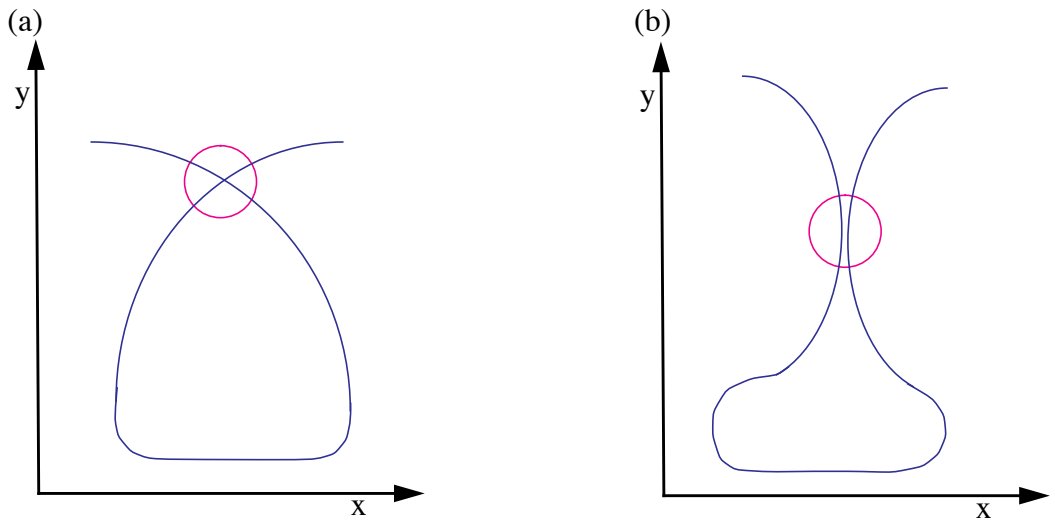

FIGURE 2. Possible cases of bifurcation points where the circle represents the actual step size. (a) The curve intersects itself; (b) The curve comes very “close” to itself.

$$\begin{aligned} &f(x_i, y_i) + f_x(x_i, y_i)dx\Delta s + f_y(x_i, y_i)dy\Delta s + \\ &f_{xx}(x_i, y_i)dx^2\Delta s^2 + 2f_{xy}(x_i, y_i)dxdy\Delta s^2 + f_{yy}(x_i, y_i)dy^2\Delta s^2 = 0 \end{aligned} \tag{9}$$

As already mentioned, the first three terms in the sum vanish, which leads to

$$f_{xx}(x_i, y_i)dx^2\Delta s^2 + 2f_{xy}(x_i, y_i)dxdy\Delta s^2 + f_{yy}(x_i, y_i)dy^2\Delta s^2 = 0 \tag{10}$$

In order to increase the readability of the following formulas, we use the following notation:

$$\begin{aligned} A &= f_{xx}(x_i, y_i) \\ B &= f_{xy}(x_i, y_i) \\ C &= f_{yy}(x_i, y_i) \end{aligned} \tag{11}$$

If we consider Equation (10) as a quadratic equation in the quotient q defined by dx/dy, we get the following solution:

$$q_{1,2} = \frac{dx}{dy} = \frac{-B \pm \sqrt{B^2 - AC}}{A} = \frac{C}{-B \mp \sqrt{B^2 - AC}} \tag{12}$$

To actually compute dx and dy, we substitute this result in Equation (3), which results in

$$\begin{pmatrix} dx \\ dy \end{pmatrix} = \pm \begin{pmatrix} \sqrt{\dfrac{1}{\dfrac{1}{q_{1,2}^2}+1}} \\ \sqrt{\dfrac{1}{q_{1,2}^2+1}} \end{pmatrix} = \pm \begin{pmatrix} \sqrt{\dfrac{C^2 - AC + 2B\,(B \pm \sqrt{B^2 - AC})}{(A-C)^2 + 4B^2}} \\ \sqrt{\dfrac{A^2 - AC + 2B\,(B \mp \sqrt{B^2 - AC})}{(A-C)^2 + 4B^2}} \end{pmatrix} \tag{13}$$

where any arbitrary combination of plus and minus from Equation (12) and Equation (13) results in four different distinct direction vectors (dx, dy). We will store all four directions in a list and follow their branches one at a time. This is explained in greater detail in Section 3.

We discussed the usual case of two intersection points and studied in detail what happens when these formulas fail when we encounter bifurcation points. Unfortunately, it could happen that we do not have four intersection points, but six, eight, or more. That means, that actually more then the two branches, as shown in Figure 1, meet at the same area. In that case, we realize that the second partial derivatives in Equation (9) vanishes also, leaving us with no choice other than including even higher order terms in the approximation of the Taylor series. This implies that we will have to solve higher order polynomials to produce the expected number of solutions.

2.3 The Corrector Step

Although we have tried hard to predict a new point p_p on the curve, it is not guaranteed to fall accurately on the curve. We can only hope to find a point that is "very close" to the curve. The goal of the *corrector step* is to find a "closest" point on the curve, starting at p_p. Among the many methods for finding zero points of functions, the Newton method is very stable and fast, and therefore very common. The Newton method is a numerical iteration method which converges to a zero point p^* of a function f, starting at a point p_0 that should be fairly close to the zero point already. In our case, the starting point for the Newton method will be p_p – the result of the predictor step.

The predictor step produced an initial value $p_0 = p_p = (x_p, y_p) = (x_0, y_0)$ for the iteration method. At each iteration step we take the new and improved value from the previous iteration and try to improve it even further, so that we actually end up on the curve itself. The result of the current iteration can be written as:

$$p_{k+1} = \begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k + \Delta x \\ y_k + \Delta y \end{pmatrix}, \tag{14}$$

where k marks the iteration number.

But how can we determine $(\Delta x, \Delta y)$? Here we exploit the same idea as we did in Euler's Method. Of course, we want that our new point (x_{k+1}, y_{k+1}) lies on the curve. So we add the condition that p_{k+1} is a curve point:

$$f(x_{k+1}, y_{k+1}) = 0. \tag{15}$$

The same ideas and manipulations as used for Equation (4) will lead us towards a solution. Substituting Equation (14) into Equation (15) and using a Taylor series expansion of f will result in:

$$f_x(x_k, y_k)\Delta x + f_y(x_k, y_k)\Delta y = -f(x_k, y_k). \tag{16}$$

That is, in fact, the formula for Newton's method for a two dimensional function. Since this is one equation with two unknowns, we have the freedom and duty to specify another condition, we want our iterate to satisfy. First of all, we want to make sure that our iteration process converges. Other than that, it would be nice if we could guarantee some sort of constant discretization of our curve. That is, we do not want the distance between samples to vary too much. Both of these ideas can be realized if we restrict our convergence to move in a right angle away from our previous iterate. To be more specific, the triangle formed by the points p_i, p_k and p_{k+1} should form a right angled triangle. Here, p_i is the initial point on the curve – the one we used in the predictor step to compute the point p_p. p_k and p_{k+1} are the previous and current iterates. In Figure 1 it was shown that this condition on our new iterate results in a somewhat circular movement around the initial point p_i. Because of this circular movement, we stay in a certain proximity to p_i, which increases the chances of convergence.

More formally, we want to add the condition that the vector $\overrightarrow{p_k p_i}$ is perpendicular to $\overrightarrow{p_k p_{k+1}}$. That is expressed in the following manner:

$$(p_i - p_k) \bullet (p_{k+1} - p_k) = \begin{bmatrix} x_i - x_k \\ y_i - y_k \end{bmatrix}^T \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \Delta x\,(x_i - x_k) + \Delta y\,(y_i - y_k) = 0 \tag{17}$$

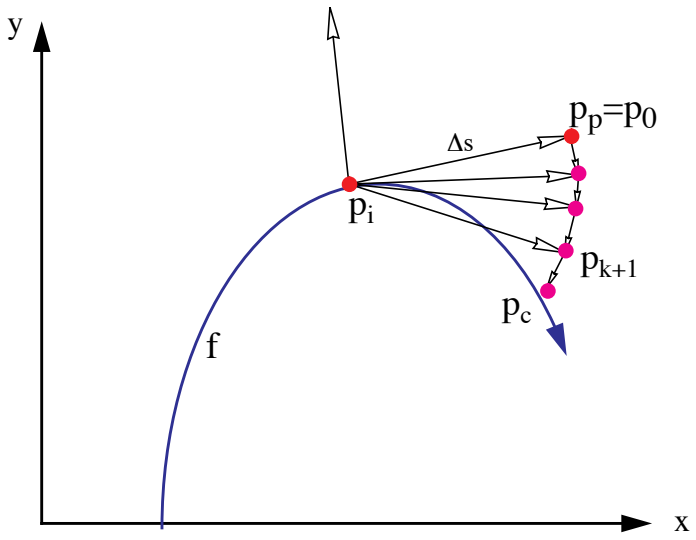

FIGURE 3. The corrector step. The Newton iterations and a forced circular movement of the iterates p_{k+1} converge towards an actual point on the curve p_c.

Equation (16) together with Equation (17) uniquely describe a new, corrected iterate p_{k+1}, which concludes the Corrector step. A geometric interpretation of these steps is illustrated in Figure 3.

3. The Algorithm

The pseudo code for our numerical algorithm is given in Figure 4. The algorithm starts by looking for a seed point (using a space subdivision method, as described in Section 3.1). The initial point will be most likely on some arbitrary part of the curve. Therefore, we will usually have two possible directions we can follow. We record both in a list for later retrieval (Line 3). At a bifurcation point a pixel will have four branches leaving it. In this case we will also record all possible branch directions in this list (Line 18). After retrieving a starting point for a new branch (Line 5) we trace the branch until we find another bifurcation point or until we reach the limits of our display device. In the case of a closed curve we also need to check when we get back to an already traced branch of the curve. A simple way to do this is explained in Section 3.5.

```
1. find an initial point p_i on the surface;
2. determine initial step size Δs;
3. record (p, Δs, bifurlist);                    /* for both initial direction */
4. repeat
5.     get p_i, Δs from bifurlist
6.     repeat                                    /* continue current branch */
7.         repeat                                /* find a reliable new point on the curve */
8.             p_p := Predictor(p_i)
9.             p_c := Corrector(p_p)
10.            C := approx_curvature(p_c, p_p)
11.            Δs := Δs/2
12.        until (p_c converges AND 1/(2C) < Δs) OR bifur_test(p_c)
13.        Δs := Δs*2;
14.        if 1/(2C) >Δs then
15.            increase Δs;
16.        draw_line(p_c, p_i);
17.        if bifur_test(p_c) then
18.            record(p_c, Δs, bifurlist)
19.        p_i := p_c;
20.    until (p_c is in the covered region or outside the screen OR bifur_test(p_c));
21. until bifurlist is empty
```

FIGURE 4. pseudo code for the curve tracking algorithm.

For each new predictor point we need to make sure that the corrector didn't fail (Line 7 through Line 12). Once we have gained confidence about our corrected point, we check whether we can adjust the step size (Line 15). We connect the previous point to the current point by a simple line (Line 16). If we found a new bifurcation point, we need to record it (Line 18) and quit this loop. In any other case we repeat this procedure and find another point on the same branch.

We now describe in more detail the implementation of some of the more involved segments of the algorithm.

3.1 Finding a Starting Point

If a seed point is not provided by the user, we employ a space subdivision approach. We divide the image region into four quadrants and compute the function at each corner of the quadrants. If the sign of the function is the same at all corners, we will apply the same subdivision to all four quadrants in a breath-first scheme. If we do find a sign change, we continue to subdivide this specific quadrant in a depth-first scheme up to the pixel- or even subpixel level, depending on how accurate our initial point should be.

3.2 Implementing the Corrector Step

Since the predicted point is only an approximation of a curve point, the corrector step is employed to converge to an actual point on the curve. Because of the approximations in our formulas, the Newton corrector step (explained in Section 2.3) usually needs to be repeated a few times. That is done within the Corrector procedure in Line 9. But how many times should we repeat the Newton step? Of course, we have to fight round off errors caused by the discrete computer arithmetic as well. The quality of our discretized curve greatly depends on our stopping criteria, as well as the successful completion of our algorithm. Therefore, we need to know when $f(p_{k+1})$ gets close enough to zero. The problem is that *close* means very different things for different functions. If we had more knowledge about the function, we could employ an absolute error test, like $f(p_{k+1}) < \varepsilon$. For a general algorithm, we will have no idea how small ε needs to be to assure spatial proximity. This notion of *being close* to the curve might also change at different ranges for the curve.

The method we chose adapts the value of ε and utilizes the fact that we draw to a finite grid. Every once in a while (e.g. every tenth time we call the Corrector procedure) we check the quality of the corrected point p_c. For this quality check we define a rectangular region centered at p_c and check for a sign change of the function f at the corner points. The size of this rectangular region can be a parameter L specified by the user or simply the current grid size. If there is a sign change in the corner points, we know that our curve will be at most $L/\sqrt{2}$ away. Therefore, we could relax our stopping criteria and set ε to $1.1|f(p_{k+1})|$. On the other hand, if we find that our corrected value isn't close enough to our curve, we should strengthen our stopping condition and set ε to $10^{-2}|f(p_{k+1})|$ and redo the corrector

step. This finally results in a relative (adaptive) error test, which is certainly superior than an absolute (static) error test.

3.3 Determining The Step Size Δs

The other crucial and very sensitive parameter of this algorithm is the step size Δs . One big advantage of our algorithm is the freedom to choose the step size. When the curve shows a small curvature we will choose a fairly large Δs , allowing the algorithm to step quickly through this “almost linear” part of the curve. On the other hand, when the curve changes rather quickly, i.e. the curvature of the function is high, we have to decrease Δs appropriately. This adaptive mechanism gives us the ability to capture the nature of our curve in greater detail when necessary.

The mathematical definition of the curvature is

$$C(p) = \lim_{q \to p} \frac{\alpha(p) - \alpha(q)}{\widehat{pq}}, \tag{18}$$

where $\alpha(p)$ denotes the angel of the tangent with the x-axis and $\widehat{pq}$ denotes the arc length between the two points. For implicit functions, C can be computed as

$$C = \frac{-f_y^2 f_{xx} + 2 f_x f_y f_{xy} - f_x^2 f_{yy}}{\left(f_x^2 + f_y^2\right)^{3/2}} \tag{19}$$

(See [6]). We compute first derivatives anyway for the Newton corrector, but would like to avoid computing second derivatives. Therefore, we use an approximation to the curvature that is based on Equation (18), where we use the current and previous point on the curve. Since we have the first derivatives already computed we can precisely compute the tangent angles. The arc length between the two points is simply approximated with the current step size. To compute the initial step size we could either use Equation (19) or just use the grid size. Choosing the initial step size for a bifurcation point can be a problem. If we are stepping too far, we might ‘loose track’ of the current branch. If we are stepping too close, the corrector might converge to a different, neighboring branch. Therefore we left it up to the user to choose this initial step size.

Another indicator of the quality of our step size would be the number of corrector steps we had to take before we converged to a curve point. In our predictor step we are basically following the tangent of the curve. If the curve is fairly linear the predicted point p_p will be almost on the curve, resulting in just very few corrector steps. If the curve changes its direction rather fast, p_p will not be the best guess and we expect more correction steps. Therefore, if we use at most one Newton step to find a point on the curve, we simply double the step size.

3.4 Detecting Bifurcation Points

Bifurcation points are points where the curve intersects itself. At these points the first derivatives of the function evaluate to zero. Of course, it will be very unlikely to find the mathematically exact bifurcation point (in case one exists), because of the numerical nature of the algorithm. Therefore, we need to keep watching the first derivatives of a function for regions where they are *close* to zero. We have seen already how difficult such a test can be (see Section 3.2). Therefore, whenever we find a sign change in one of the partial derivatives (from the previous to the current point) we have good reason to suspect that there might be a bifurcation point close by. As a result, we decrease our step size and repeat the last predictor/corrector step until the step size is smaller than a given size, which can be defined by the user. If there is still reason to believe that there might be a bifurcation point, we create a rectangular region around the current point on the curve. Similar to the test we performed in Section 3.2, we test whether there is a sign change in the partial derivatives f_x and f_y. If we find a sign change in both partial derivatives we conclude that we detected a bifurcation point. Otherwise, we conclude that we probably found a turning point of the curve.

3.5 Stopping Criteria

We assume that we deal with a closed curve. Therefore, we need to find out if and when we returned to a part of a branch we have traced already. If we do not include this exit test (in Line 20), we will trace the curve over and over again, ending up in an infinite loop.

Because of the nature of a numeric algorithm it is very unlikely to return to the exact same point from where we started. Therefore, we cannot just test for equality. We also do not want to test the proximity

of the new point p_c to every one of the computed values along the curve. That might be possible for the first few values, when the cost is still low, but certainly not for arbitrary curves, where we have to compute many points to reasonably approximate the curves shape. The other problem is again the meaning of "very close". To solve this problem, we maintain a counter that counts the number of times we attempt to draw a pixel that has been drawn already. When the algorithm computes a point on the curve that is mapped to a pixel that was not drawn before, we set the counter to zero. Every time we compute a new point on the curve within a pixel that was previously drawn, we increase a counter by one. If the counter's value exceeds some threshold, the algorithm stops pursuing this path.

TABLE 1. Performance results comparing our algorithm. The empty cells denote cases where Chandler's algorithm could not draw the curve.

Figure	# Function evaluation per pixel		# pixels per predictor	# pixels per predictor
	Our Method	Chandler		
5a	1.89	2.68	6.21	2.55
5b	2.01	2.80	6.79	3.20
5c	3.27	2.92	3.51	2.49
6a	2.88	–	6.66	5.04
6b	4.18	–	4.05	3.67
6c	4.55	–	2.38	2.30

4. Results

We applied our algorithm to many different functions with and without bifurcation points and found that they were drawn correctly in any image resolution of 128^2 to 512^2. For images of low resolution (32^2 and sometimes 64^2) the user may have to explicitly specify a smaller step size. We have chosen three very common functions without a bifurcation point (Figure 5) and three interesting functions with many bifurcation points (Figure 6). Some interesting statistics for each group of functions is summarized in Table 1 and compared to Chandlers algorithm [7]. The test cases where drawn onto a 5122 image. Drawing times, for all examples, was under 0.2 seconds on a Silicon Graphics Crimson (150Mhz R4400). In the table we also provide the average number of pixels per one predictor step (showing the adaptive step size), and the average number of correct steps per predictor.

In general, the number of function evaluations per pixel for low resolutions is expectably high, since the curvature of a curve considered within a pixel is larger than if the same curve is subdivided into smaller parts (i.e. for smaller pixel sizes). Therefore, the step size (in terms of the number of pixels) is smaller and the algorithm must perform more calculations on a per pixel basis. For high resolutions we have much fewer function evaluations per pixel than Chandler's algorithm without loss of quality. This demonstrates the power of the adaptive step size.

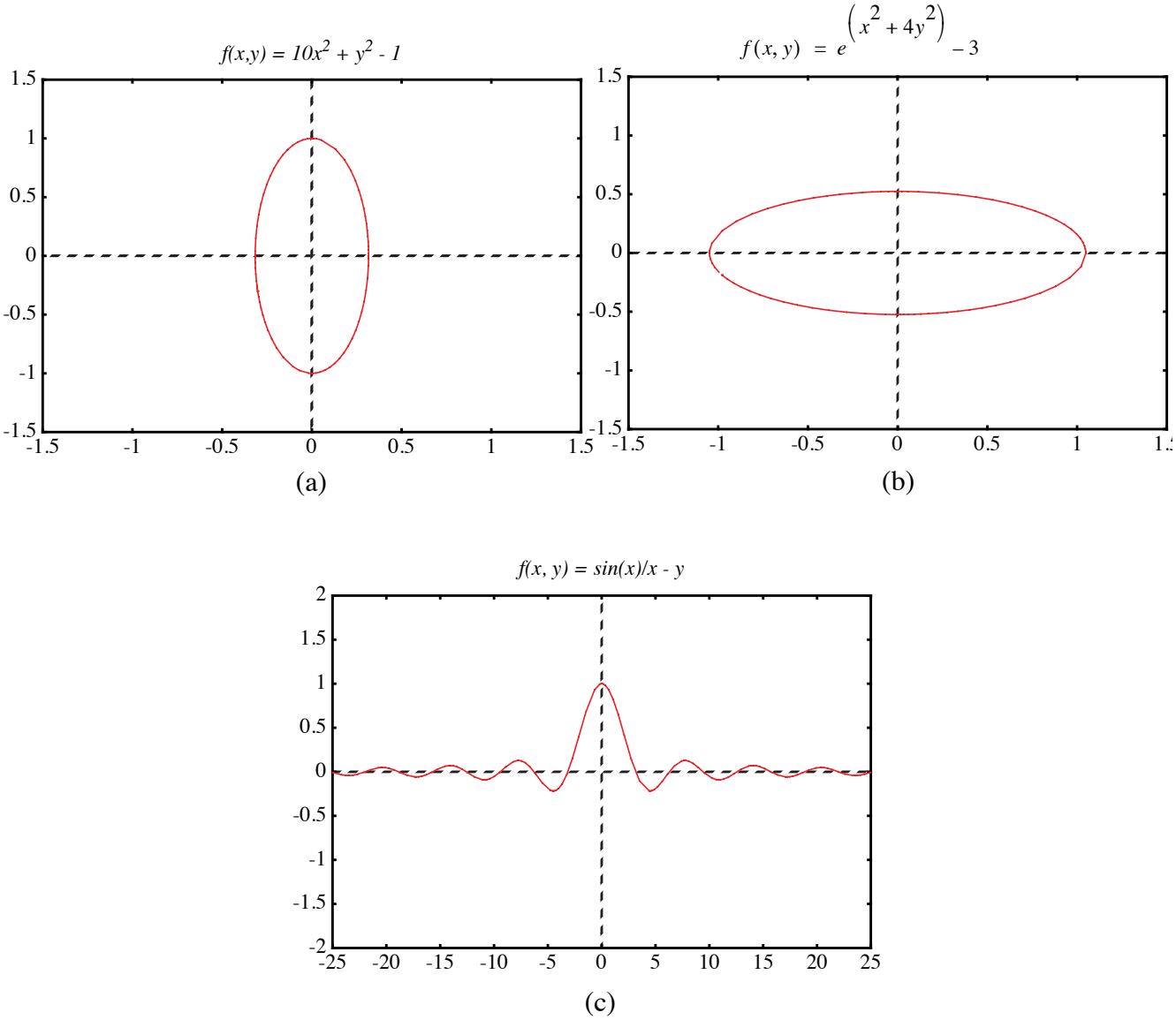

FIGURE 5. Curves without bifurcation points.

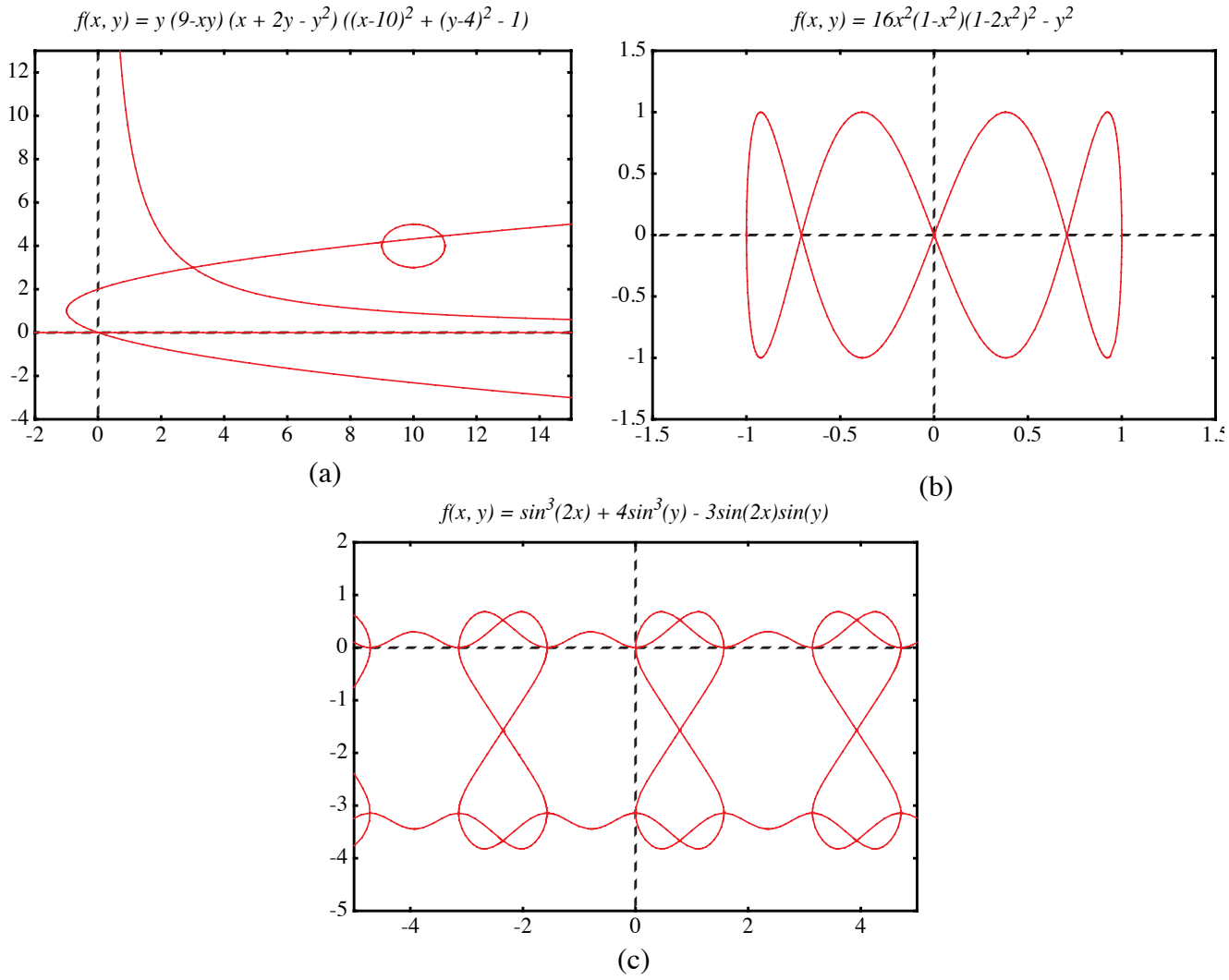

FIGURE 6. Curves with bifurcation points.

5. Conclusions

We presented here a numerical algorithm that is able to draw a large family of implicit functions. Through an adaptive step size design our performance is greatly optimized. We have shown that our approach outperforms previous tracking algorithms.

Our future research includes the improvement of the accuracy of the predictor step. Since we already know the derivatives at the previous point and, most of the time, also have this information available for the point before that, we could use the Hermite interpolation scheme and approximate our curve with a

polynom of third degree. This higher order approximation of our curve should result in fewer corrector steps. Since the higher order polynomial is computed only at the predictor step, saving several corrector steps, we expect a more efficient algorithm. A similar improvement can be made when connecting a new point on the curve with the previous one. Instead of a line through these point, one could use a higher order polynomial. Improvements to the accuracy of the corrector procedure are also possible, for example, by using a better corrector than Newton's. Finally, we plan to explore the extension of our methods to 3D voxelization of implicit surfaces.

6. References

[1] Allgower E. and S. Gnutzmann, "Simplicial Pivoting for Mesh Generation of Implicitly Defined Surfaces", Computer Aided Geometric Design, 8(4), 1991, pp. 30.

[2] Arvo J. and K. Novins, "Iso-Contour Volume Rendering", Proceedings of 1994 Symposium On Volume Visualization.

[3] Baker G. and E. Overman, "The Art of Scientific Computing", Draft Edition, The Ohio State Bookstore.

[4] Blinn J.F., "A Generalization of Algebraic Surface Drawing", ACM Transactions on Graphics, 1(3), 1982, pp. 235-256.

[5] Bloomenthal J., "Polygonization of Implicit Surfaces", Computer Aided Geometric Design, 5(4), 1988, pp. 341-356.

[6] Bronstein I. N. and Semendjajew, K. A. "Taschenbuch der Mathematik", BSB B.G.Teubner Verlagsgesellschaft, Leipzig, 1985.

[7] Chandler R.E., "A Tracking Algorithm for Implicitly Defined Curves", IEEE Computer Graphics and Applications, 8(2), March 1988, pp. 83-89.

[8] Foley J. D., A. van Dam, S. K. Feiner and J. F. Hughes, "Computer Graphics Principles and Practice", Addison-Wesley, 1990.

[9] Jordan B.W., W.J. Lennon, and B.D. Holm, "An Improved Algorithm for the Generation of Nonparametric Curves", IEEE Trans. Computers, Vol. C-22, 1973, pp. 1052-1060.

[10] Hall M. and J.Warren, "Adaptive Polygonization of Implicitly Defined Surfaces", IEEE Computer Graphics and Applications, 10(6), November 1990, pp. 33-42.

[11] Hanrahan P., "Ray Tracing Algebraic Surfaces", Computer Graphics (SIGGRAPH'83 Proceedings), 17(3), 1983, pp. 83-90.

[12] J.C. Hart. Ray Tracing Implicit Surfaces. WSU Technical Report EECS-93-014. Chapter in "Design, Visualization, and Animation of Implicit Surfaces," ACM SIGGRAPH '93 intermediate course notes, J. Bloomenthal and B. Wyvill, eds., Aug. 1993.

[13] Hobby J.D., "Rasterization of Nonparametric Curves", ACM Transactions on Graphics, 9(3), July 1990, pp. 262-277.

[14] Kalra D. and A.H. Barr, "Guaranteed Ray Intersections with Implicit Surfaces", Computer Graphics (SIGGRAPH'89 Proceedings), 23, 1989, pp. 297-306.

[15] Lorensen W. and H. Cline, “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”, Computer Graphics, 21(4), 1987, pp. 163-169.

[16] Sederberg T.W., D.C.Anderson, and R.N.Goldman, “Implicitization, Inversion, and Intersection of Planar Rational Cubic Curves”, Computer Vision, Graphics, and Image Processing, 31(1), 1985, pp. 89-102.

[17] Sederberg T.W. and A.K.Zundel, “Scan Line Display of Algebraic Surfaces”, Computer Graphics (SIGGRAPH’89 Proceedings), 23(4), 1989, pp. 147-156.

Efficient Rasterization of Implicit Functions

Torsten Möller and Roni Yagel

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio

{moeller, yagel}@cis.ohio-state.edu

Abstract

Implicit curves are widely used in computer graphics because of their powerful features for modeling and their ability for general function description. The most popular rasterization techniques for implicit curves are space subdivision and curve tracking. In this paper we are introducing an efficient curve tracking algorithm that is also more robust then existing methods. We employ the Predictor-Corrector Method on the implicit function to get a very accurate curve approximation in a short time. Speedup is achieved by adapting the step size to the curvature. In addition, we provide mechanisms to detect and properly handle bifurcation points, where the curve intersects itself. Finally, the algorithm allows the user to trade-off accuracy for speed and vice a versa. We conclude by providing examples that demonstrate the capabilities of our algorithm.

1. Introduction

Implicit surfaces are surfaces that are defined by implicit functions. An implicit function f is a mapping from an n dimensional space R^n to the space R, described by an expression in which all variables appear on one side of the equation. This is a very general way of describing a mapping. An n dimensional implicit surface S is defined by all the points p in R^n such that the implicit mapping f projects p to 0. More formally, one can describe S by

$$S_f = \{p \mid p \in R^n, f(p) = 0\} . \quad (1)$$

Often times, one is interested in visualizing *isosurfaces* (or *isocontours* for 2D functions) which are functions of the form $f(p) = c$ for some constant c.

Different classes of implicit surfaces are of importance. Implicit surfaces that are defined by a polynomial are called *algebraic surfaces*. Since almost all continuous functions can be approximated by polynomials, algebraic surfaces represent a powerful class. Since polynomials have been studied and understood fairly well, one has a better mathematical handle at algebraic surfaces.

Another special class of implicit surfaces have density functions as their defining functions. Density functions decrease exponentially with increasing distance to a certain center point. The models where these functions find usage usually arise in the study of physical phenomena, e.g. atom models. Blinn [4] was one of the first to try to visualize these surfaces on a computer. They became to be known as Blinn's "*Blobby Model*" or *Metaballs*.

Yet another model which is gaining importance is the *discrete data set*. Such data set is commonly generated by various medical scanners such as MRI and CT. These data sets can also be seen as implicit functions, where $f(p) = v$ and v is the scanned value at the location p. We assume that if p is not on a grid point then some sort of interpolation from neighboring grid points defines v at p.

Rendering implicit surfaces is very difficult and computationaly very expensive. Usually existing algorithms to render implicit surfaces work only for special cases or special sets of functions. In fact, to date there is no such algorithm that can render arbitrary implicit surfaces accurately and in a stable and timely manner.

Algorithms for rendering 3D implicit surfaces can be classified into three groups: representation transforms, ray tracing, and surface tracking. The first approach tries to transform the problem from implicit surfaces to another representation which is easier to display such as polygon meshes [1][5][10][15] or parametric representations [16]. The second approach to rendering of implicit surfaces is to ray-trace them. Efficient ray-object intersection algorithms have been devised for algebraic surfaces [11], metaballs [4], and a class of functions for which Lipschitz constants can be efficiently computed [12][14]. The third approach to rendering implicit functions is to scan convert or discretize the function. Seder-

berg and Zundel [17] introduced such a scan line algorithm for a class of implicit functions that can be represented by Bernstein polynomials.

When it comes to drawing 2D implicit functions (curves) an attractive approach is to follow the path of the curve. The algorithm starts at a seed point on the curve. It then looks at the pixels adjacent to the seed point to see if any of them is the next pixel along the curve. Various algorithms exist which mainly differ in the method they use to find the next pixel along the curve.

One way to find the next pixel is based on *digital difference analysis* (DDA). A lot of work has been done in this field and the most popular algorithm applying this technique are Bresenham's and the midpoint line and circle drawing algorithms [8]. The rate of change in x and y is computed through fast integral updates of dx and dy, which are then used as decision variables to determine the next pixel center. This idea was generalized for algebraic curves by Hobby [13]. One problem of this approach is that the inherent integer arithmetic can only be exploited for a small set of algebraic functions. Furthermore, this algorithm is not able to deal properly with self intersecting curves. Hobby assumes that these intersection points are given by the user.

A different way to continue a curve would be to evaluate the underlying function at the center of the neighboring pixels to find the one with the smallest absolute function value. This idea has been implemented by Jordan et. al. [9] and was improved by Chandler [7]. Since there are only eight neighboring pixels, eight function evaluations are the maximum needed to find the next pixel on the curve. Since we will very likely be traveling in the same direction we have been traveling when we picked the current pixel, Chandler tests first the pixel continuing in the same direction, then its neighbors etc. For most 'nicely' behaving functions two to three function evaluations per pixel will be sufficient. To increase accuracy, Chandler prefers to look for a sign change rather than the smallest absolute value. Therefore, he evaluates the function not at pixel centers but rather halfway in between pixels. Although the algorithm does not depend on the actual description of the function and therefore is applicable to arbitrary functions, it is still not able to deal with self intersecting functions. To deal with such points the algorithm requires user intervention. Alternatively, it includes a brute-force Monte Carlo method to visualize the positive and negative regions of the function. Finally, if the curve enters and leaves the pixel in-

between the same two neighboring sample points, this algorithm will not be able to register a sign change and therefore fail.

For density functions, Arvo et. al. applied a numerical method known as the predictor-corrector method [2]. Their main goal is the rendering of iso-curves of medical data sets. In this paper we introduce an efficient and more robust predictor-corrector method. Our algorithm deals with self intersecting curves properly and increases the efficiency of the algorithm to less than one function evaluation per pixel on the average. Finally, unlike existing curve tracking methods, our algorithm can be applied to a large family of implicit functions including selfintersecting functions with at most two branches at the intersection point.

2. The Predictor-Corrector Method

The predictor-corrector method is a well known numerical algorithm which has its origin in solving ordinary differential equations and bifurcation theory. Arvo and Novins [2] used a two dimensional standard method for extracting isocurves in a 2D medical image. Although the predictor-corrector method offers ways to detect and properly deal with self intersecting curves and curves “very close” to themselves, Arvo and Novins did not explore these capabilities.

The predictor-corrector method is a numerical continuation method (tracking algorithm). The goal is to find a point set which represents a discretized approximation of a curve that is defined by some implicit function as in Equation (1). As the name suggests, the predictor-corrector method consists of two steps. In the predictor step we take a (more or less sophisticated) guess of a point on the curve, denoted by p_p. Depending on how much thought and work we put into predicting the point p_p, it will be closer or farther away from an actual point on the curve. Depending on the assumption we make about the curve and its actual shape, we will not be able to always accurately predict an exact point on the curve. In the corrector step we use our predicted point p_p as a starting point for an iterative method that converges to an actual point on the curve. We denote this new, corrected point, by p_c. We repeat these two steps until we tracked the whole curve. The resulting discrete point set can then be displayed on a screen.

There are many different ways to implement the predictor or corrector steps. We use the standard Euler-Newton method with an adaptive step size and introduce techniques to handle selfintersecting curves.

2.1 The Predictor Step

The focus of the predictor step is to find a point p_p that is very close or even on the curve, so that the costs for the corrector step are reduced or even eliminated. If we already know points on the curve, we can approximate the curve through a polynomial and get a new approximate for this curve through extrapolation or interpolation.

Finding the very first point on the curve is not always easy. One usually has some understanding of the kind of function one tries to discretize and can therefore provide an initial seed value to the algorithm. If this is not possible, one needs to apply a brute force algorithm or some other heuristic to find an initial seed value. The image space is limited (usually a screen in computer graphics). Therefore one rather brute-force method would be to impose a grid onto the screen and evaluate the underlying implicit function at each grid point to choose a closest point (where $|f(p)|$ becomes smallest). Another possibility would be to look for a sign change of the underlying function between neighboring grid points. We employ a faster method that is based on a quadtree subdivision algorithm as explained in Section 3.1.

We now present one way to implement the predictor step. *One-step methods* only employ one point (the point of the previous iteration) to predict a new point. The standard Euler method is a good example of a one-step method. However, one can envision the use of Hermites method which employs more than one point and is therefore called *multi-step method*. Although multi-step methods are usually more expensive, they lead to better, higher order approximations, reducing the number of necessary corrector steps.

Euler's method starts at a point which is known to be on the curve. Because this is a one-step method, each point on the curve will lead to exactly one new point on the curve. Since we execute many predictor-corrector iterations to obtain a point set that approximates our curve, it is just obvious to use the resulting point of the previous iteration as the initial point for the current iteration. We call this initial point $p_i = (x_i, y_i)$. We also want to have some control over the distance between the new point and

the initial point. That is, we want p_p to be located at a certain distance Δs from p_i and not at some unpredictable "end" of the curve. Δs is called the arc length parameter and it represents a parameterization of our curve. The linear approximation of the predicted point results in

$$\begin{aligned} x_p &= x_i + dx\Delta s \\ y_p &= y_i + dy\Delta s \end{aligned}, \tag{2}$$

where $p_p = (x_p, y_p)$ now depends on the arc length and the initial point. Since Δs is the euclidean distance between p_p and p_i, we need to make sure that:

$$dx^2 + dy^2 = 1 \tag{3}$$

Equation (3) defines a unit circle around the origin. Combining it with Equation (2) we are describing a circle of radius Δs and origin p_i. Next, we have to find the intersection points of that circle with our curve. We choose one of these points as our new predicted value. Since the origin of this circle lies already on the curve, we are guaranteed that our circle will intersect our implicit curve, unless the whole curve is contained within the circle. In that particular case, the choice of the distance Δs between neighboring points is not appropriate and needs to be changed. If Δs is in the order of the size of a pixel and the curve is still contained within such a small circle, the discretization of this curve would just be one pixel and we are done.

Assuming a closed curve, we will actually get an even number of intersection points. Choosing our step size (arc length) small enough, we will get exactly two intersection points (see Figure 1). However, in the case of a self intersecting curve or when a curve comes "close" to itself, we will get more than two intersection points, even if Δs is very small. We will explore these special cases in Section 2.2.

The second condition for the new predicted point is, of course, that it should lie on the curve defined by the underlying implicit function *f*. That means we require

$$f(x_p, y_p) = 0\,. \tag{4}$$

Since we are processing an arbitrary functions *f* we are not able to use Equation (4) directly to compute p_p. Instead, we substitute Equation (2) into Equation (4) and through a Taylor series expansion we are

able to compute an approximation to the actual point on the curve. This approximation serves as the predicted value p_p.

Let's have a look at the equations. The substitution results in

$$f(x_i + dx\Delta s, y_i + dy\Delta s) = 0. \tag{5}$$

Now the actual approximation takes place. We expand Equation (5) in a Taylor series and keep only the linear terms. That results in:

$$f(x_i, y_i) + \frac{\partial}{\partial x} f(x_i, y_i) dx\Delta s + \frac{\partial}{\partial y} f(x_i, y_i) dy\Delta s = 0 \tag{6}$$

The initial point was assumed to be on the curve, i.e. $f(x_i, y_i) = 0$. We conclude

$$\frac{\partial}{\partial x} f(x_i, y_i) dx\Delta s + \frac{\partial}{\partial y} f(x_i, y_i) dy\Delta s = 0 \tag{7}$$

Using Equation (3) and Equation (7), we can actually solve for the unknowns (dx, dy). Using a short-cut notation for the partial derivatives of $f(x_i, y_i)$, where $\frac{\partial}{\partial x} f(x_i, y_i)$ is replaced by f_x (and similarly for the partial derivative in y) the result can be written as

$$\begin{aligned} dx &= \delta \frac{f_y}{\sqrt{f_x^2 + f_y^2}} \\ dy &= -\delta \frac{f_x}{\sqrt{f_x^2 + f_y^2}} \end{aligned} \tag{8}$$

which is also proven by Baker and Overman [3]. Here δ stands for the sign plus or minus which represents the direction in which we continue to track the curve. In the case of the seed point we want to make sure, that we trace both directions. However, in the general case we don't want to step back into the direction we were coming from. A simple sign check of the scalar product of the direction, where we were coming from and our new direction, helps us determining the appropriate δ.

Although our new predicted value lies on a circle around the initial point, we only find a point near the curve because of the approximation in Equation (6). If we take a closer look at Equation (7), the one we actually used as an approximation to Equation (5), we realize that (dx, dy) describes the components

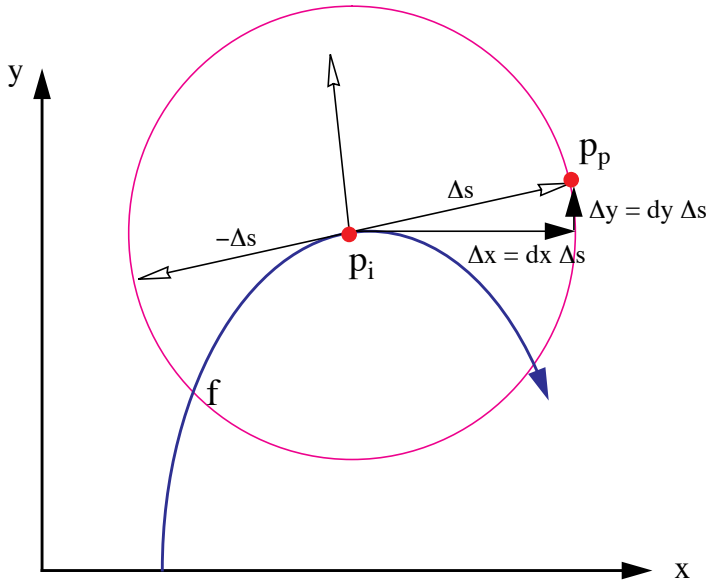

FIGURE 1. The predictor step. Here Δs is the step size we want to step along the curve starting at the initial point p_i. Because Equation (7) is only an approximation, the predicted value will not necessarily lie on the curve described by f.

of the tangent to the curve at the initial point. That leads us to a better geometric understanding of this process, captured in Figure 1.

2.2 Bifurcation Points

The intersection of the circle equation (originating in Equation (3)) with the implicit curve results usually in two points. As pointed out earlier, in case of a selfintersecting curve (Figure 2a) or if it just comes "close" to itself (Figure 2b) we expect at least four intersection points. These special points are also called *bifurcation points*. Looking at our formulas in Equation (7) and Equation (3), we are only able to compute two points, since it is an intersection of a quadratic (Equation (3)) curve with a linear curve (Equation (7)). In a case of a bifurcation point the linear approximation of the Taylor series as in Equation (7) will not be enough. In fact, from the implicit function theorem we can conclude, that the first partial derivatives will be zero for these bifurcation points. Therefore we cannot compute (dx, dy) as derived above. We have to include the second order terms of the Taylor series expansion in Equation (6) to get the appropriate results.

The more accurate approximation to Equation (5), replacing Equation (6), is now

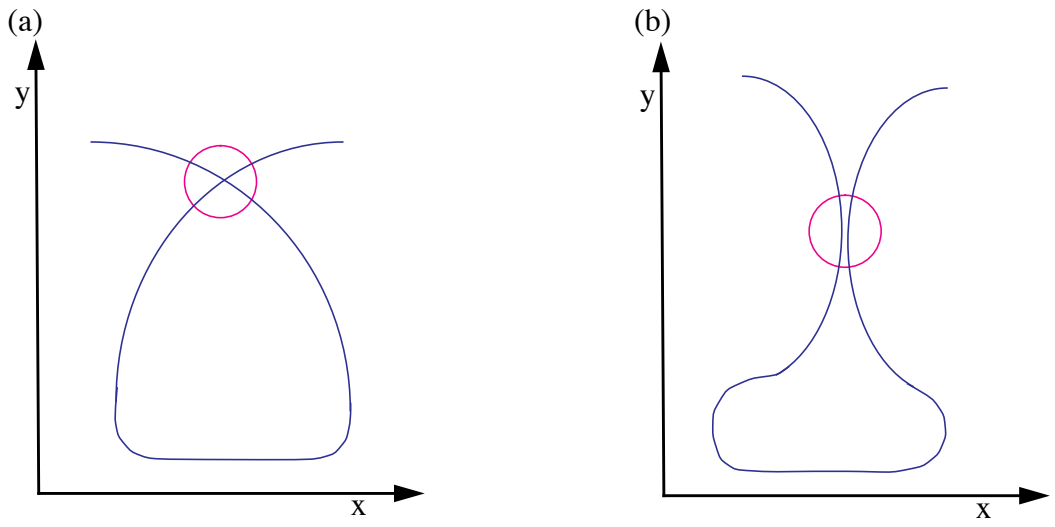

FIGURE 2. Possible cases of bifurcation points where the circle represents the actual step size. (a) The curve intersects itself; (b) The curve comes very “close” to itself.

$$\begin{aligned} &f(x_i, y_i) + f_x(x_i, y_i)dx\Delta s + f_y(x_i, y_i)dy\Delta s + \\ &f_{xx}(x_i, y_i)dx^2\Delta s^2 + 2f_{xy}(x_i, y_i)dxdy\Delta s^2 + f_{yy}(x_i, y_i)dy^2\Delta s^2 = 0 \end{aligned} \tag{9}$$

As already mentioned, the first three terms in the sum vanish, which leads to

$$f_{xx}(x_i, y_i)dx^2\Delta s^2 + 2f_{xy}(x_i, y_i)dxdy\Delta s^2 + f_{yy}(x_i, y_i)dy^2\Delta s^2 = 0 \tag{10}$$

In order to increase the readability of the following formulas, we use the following notation:

$$\begin{aligned} A &= f_{xx}(x_i, y_i) \\ B &= f_{xy}(x_i, y_i) \\ C &= f_{yy}(x_i, y_i) \end{aligned} \tag{11}$$

If we consider Equation (10) as a quadratic equation in the quotient q defined by dx/dy, we get the following solution:

$$q_{1,2} = \frac{dx}{dy} = \frac{-B \pm \sqrt{B^2 - AC}}{A} = \frac{C}{-B \mp \sqrt{B^2 - AC}} \tag{12}$$

To actually compute dx and dy, we substitute this result in Equation (3), which results in

$$\begin{pmatrix} dx \\ dy \end{pmatrix} = \pm \begin{pmatrix} \sqrt{\dfrac{1}{\dfrac{1}{q_{1,2}^2}+1}} \\ \sqrt{\dfrac{1}{q_{1,2}^2+1}} \end{pmatrix} = \pm \begin{pmatrix} \sqrt{\dfrac{C^2 - AC + 2B\,(B \pm \sqrt{B^2 - AC})}{(A-C)^2 + 4B^2}} \\ \sqrt{\dfrac{A^2 - AC + 2B\,(B \mp \sqrt{B^2 - AC})}{(A-C)^2 + 4B^2}} \end{pmatrix} \tag{13}$$

where any arbitrary combination of plus and minus from Equation (12) and Equation (13) results in four different distinct direction vectors (dx, dy). We will store all four directions in a list and follow their branches one at a time. This is explained in greater detail in Section 3.

We discussed the usual case of two intersection points and studied in detail what happens when these formulas fail when we encounter bifurcation points. Unfortunately, it could happen that we do not have four intersection points, but six, eight, or more. That means, that actually more then the two branches, as shown in Figure 1, meet at the same area. In that case, we realize that the second partial derivatives in Equation (9) vanishes also, leaving us with no choice other than including even higher order terms in the approximation of the Taylor series. This implies that we will have to solve higher order polynomials to produce the expected number of solutions.

2.3 The Corrector Step

Although we have tried hard to predict a new point p_p on the curve, it is not guaranteed to fall accurately on the curve. We can only hope to find a point that is "very close" to the curve. The goal of the *corrector step* is to find a "closest" point on the curve, starting at p_p. Among the many methods for finding zero points of functions, the Newton method is very stable and fast, and therefore very common. The Newton method is a numerical iteration method which converges to a zero point p^* of a function f, starting at a point p_0 that should be fairly close to the zero point already. In our case, the starting point for the Newton method will be p_p – the result of the predictor step.

The predictor step produced an initial value $p_0 = p_p = (x_p, y_p) = (x_0, y_0)$ for the iteration method. At each iteration step we take the new and improved value from the previous iteration and try to improve it even further, so that we actually end up on the curve itself. The result of the current iteration can be written as:

$$p_{k+1} = \begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k + \Delta x \\ y_k + \Delta y \end{pmatrix}, \tag{14}$$

where k marks the iteration number.

But how can we determine $(\Delta x, \Delta y)$? Here we exploit the same idea as we did in Euler's Method. Of course, we want that our new point (x_{k+1}, y_{k+1}) lies on the curve. So we add the condition that p_{k+1} is a curve point:

$$f(x_{k+1}, y_{k+1}) = 0. \tag{15}$$

The same ideas and manipulations as used for Equation (4) will lead us towards a solution. Substituting Equation (14) into Equation (15) and using a Taylor series expansion of f will result in:

$$f_x(x_k, y_k)\Delta x + f_y(x_k, y_k)\Delta y = -f(x_k, y_k). \tag{16}$$

That is, in fact, the formula for Newton's method for a two dimensional function. Since this is one equation with two unknowns, we have the freedom and duty to specify another condition, we want our iterate to satisfy. First of all, we want to make sure that our iteration process converges. Other than that, it would be nice if we could guarantee some sort of constant discretization of our curve. That is, we do not want the distance between samples to vary too much. Both of these ideas can be realized if we restrict our convergence to move in a right angle away from our previous iterate. To be more specific, the triangle formed by the points p_i, p_k and p_{k+1} should form a right angled triangle. Here, p_i is the initial point on the curve – the one we used in the predictor step to compute the point p_p. p_k and p_{k+1} are the previous and current iterates. In Figure 1 it was shown that this condition on our new iterate results in a somewhat circular movement around the initial point p_i. Because of this circular movement, we stay in a certain proximity to p_i, which increases the chances of convergence.

More formally, we want to add the condition that the vector $\overrightarrow{p_k p_i}$ is perpendicular to $\overrightarrow{p_k p_{k+1}}$. That is expressed in the following manner:

$$(p_i - p_k) \bullet (p_{k+1} - p_k) = \begin{bmatrix} x_i - x_k \\ y_i - y_k \end{bmatrix}^T \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \Delta x\,(x_i - x_k) + \Delta y\,(y_i - y_k) = 0 \tag{17}$$

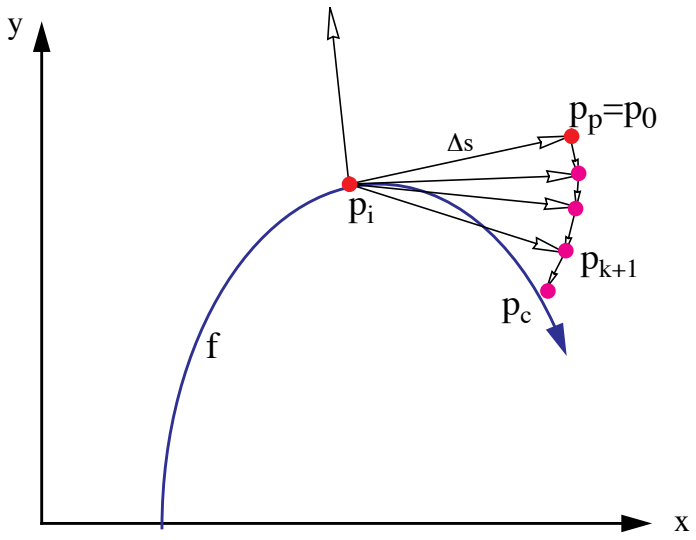

FIGURE 3. The corrector step. The Newton iterations and a forced circular movement of the iterates p_{k+1} converge towards an actual point on the curve p_c.

Equation (16) together with Equation (17) uniquely describe a new, corrected iterate p_{k+1}, which concludes the Corrector step. A geometric interpretation of these steps is illustrated in Figure 3.

3. The Algorithm

The pseudo code for our numerical algorithm is given in Figure 4. The algorithm starts by looking for a seed point (using a space subdivision method, as described in Section 3.1). The initial point will be most likely on some arbitrary part of the curve. Therefore, we will usually have two possible directions we can follow. We record both in a list for later retrieval (Line 3). At a bifurcation point a pixel will have four branches leaving it. In this case we will also record all possible branch directions in this list (Line 18). After retrieving a starting point for a new branch (Line 5) we trace the branch until we find another bifurcation point or until we reach the limits of our display device. In the case of a closed curve we also need to check when we get back to an already traced branch of the curve. A simple way to do this is explained in Section 3.5.

```
1. find an initial point p_i on the surface;
2. determine initial step size Δs;
3. record (p, Δs, bifurlist);                    /* for both initial direction */
4. repeat
5.     get p_i, Δs from bifurlist
6.     repeat                                    /* continue current branch */
7.         repeat                                /* find a reliable new point on the curve */
8.             p_p := Predictor(p_i)
9.             p_c := Corrector(p_p)
10.            C := approx_curvature(p_c, p_p)
11.            Δs := Δs/2
12.        until (p_c converges AND 1/(2C) < Δs) OR bifur_test(p_c)
13.        Δs := Δs*2;
14.        if 1/(2C) >Δs then
15.            increase Δs;
16.        draw_line(p_c, p_i);
17.        if bifur_test(p_c) then
18.            record(p_c, Δs, bifurlist)
19.        p_i := p_c;
20.    until (p_c is in the covered region or outside the screen OR bifur_test(p_c));
21. until bifurlist is empty
```

FIGURE 4. pseudo code for the curve tracking algorithm.

For each new predictor point we need to make sure that the corrector didn't fail (Line 7 through Line 12). Once we have gained confidence about our corrected point, we check whether we can adjust the step size (Line 15). We connect the previous point to the current point by a simple line (Line 16). If we found a new bifurcation point, we need to record it (Line 18) and quit this loop. In any other case we repeat this procedure and find another point on the same branch.

We now describe in more detail the implementation of some of the more involved segments of the algorithm.

3.1 Finding a Starting Point

If a seed point is not provided by the user, we employ a space subdivision approach. We divide the image region into four quadrants and compute the function at each corner of the quadrants. If the sign of the function is the same at all corners, we will apply the same subdivision to all four quadrants in a breath-first scheme. If we do find a sign change, we continue to subdivide this specific quadrant in a depth-first scheme up to the pixel- or even subpixel level, depending on how accurate our initial point should be.

3.2 Implementing the Corrector Step

Since the predicted point is only an approximation of a curve point, the corrector step is employed to converge to an actual point on the curve. Because of the approximations in our formulas, the Newton corrector step (explained in Section 2.3) usually needs to be repeated a few times. That is done within the Corrector procedure in Line 9. But how many times should we repeat the Newton step? Of course, we have to fight round off errors caused by the discrete computer arithmetic as well. The quality of our discretized curve greatly depends on our stopping criteria, as well as the successful completion of our algorithm. Therefore, we need to know when $f(p_{k+1})$ gets close enough to zero. The problem is that *close* means very different things for different functions. If we had more knowledge about the function, we could employ an absolute error test, like $f(p_{k+1}) < \varepsilon$. For a general algorithm, we will have no idea how small ε needs to be to assure spatial proximity. This notion of *being close* to the curve might also change at different ranges for the curve.

The method we chose adapts the value of ε and utilizes the fact that we draw to a finite grid. Every once in a while (e.g. every tenth time we call the Corrector procedure) we check the quality of the corrected point p_c. For this quality check we define a rectangular region centered at p_c and check for a sign change of the function f at the corner points. The size of this rectangular region can be a parameter L specified by the user or simply the current grid size. If there is a sign change in the corner points, we know that our curve will be at most $L/\sqrt{2}$ away. Therefore, we could relax our stopping criteria and set ε to $1.1|f(p_{k+1})|$. On the other hand, if we find that our corrected value isn't close enough to our curve, we should strengthen our stopping condition and set ε to $10^{-2}|f(p_{k+1})|$ and redo the corrector

step. This finally results in a relative (adaptive) error test, which is certainly superior than an absolute (static) error test.

3.3 Determining The Step Size Δs

The other crucial and very sensitive parameter of this algorithm is the step size Δs . One big advantage of our algorithm is the freedom to choose the step size. When the curve shows a small curvature we will choose a fairly large Δs , allowing the algorithm to step quickly through this “almost linear” part of the curve. On the other hand, when the curve changes rather quickly, i.e. the curvature of the function is high, we have to decrease Δs appropriately. This adaptive mechanism gives us the ability to capture the nature of our curve in greater detail when necessary.

The mathematical definition of the curvature is

$$C(p) = \lim_{q \to p} \frac{\alpha(p) - \alpha(q)}{\widehat{pq}}, \tag{18}$$

where $\alpha(p)$ denotes the angel of the tangent with the x-axis and $\widehat{pq}$ denotes the arc length between the two points. For implicit functions, C can be computed as

$$C = \frac{-f_y^2 f_{xx} + 2 f_x f_y f_{xy} - f_x^2 f_{yy}}{\left(f_x^2 + f_y^2\right)^{3/2}} \tag{19}$$

(See [6]). We compute first derivatives anyway for the Newton corrector, but would like to avoid computing second derivatives. Therefore, we use an approximation to the curvature that is based on Equation (18), where we use the current and previous point on the curve. Since we have the first derivatives already computed we can precisely compute the tangent angles. The arc length between the two points is simply approximated with the current step size. To compute the initial step size we could either use Equation (19) or just use the grid size. Choosing the initial step size for a bifurcation point can be a problem. If we are stepping too far, we might ‘loose track’ of the current branch. If we are stepping too close, the corrector might converge to a different, neighboring branch. Therefore we left it up to the user to choose this initial step size.

Another indicator of the quality of our step size would be the number of corrector steps we had to take before we converged to a curve point. In our predictor step we are basically following the tangent of the curve. If the curve is fairly linear the predicted point p_p will be almost on the curve, resulting in just very few corrector steps. If the curve changes its direction rather fast, p_p will not be the best guess and we expect more correction steps. Therefore, if we use at most one Newton step to find a point on the curve, we simply double the step size.

3.4 Detecting Bifurcation Points

Bifurcation points are points where the curve intersects itself. At these points the first derivatives of the function evaluate to zero. Of course, it will be very unlikely to find the mathematically exact bifurcation point (in case one exists), because of the numerical nature of the algorithm. Therefore, we need to keep watching the first derivatives of a function for regions where they are *close* to zero. We have seen already how difficult such a test can be (see Section 3.2). Therefore, whenever we find a sign change in one of the partial derivatives (from the previous to the current point) we have good reason to suspect that there might be a bifurcation point close by. As a result, we decrease our step size and repeat the last predictor/corrector step until the step size is smaller than a given size, which can be defined by the user. If there is still reason to believe that there might be a bifurcation point, we create a rectangular region around the current point on the curve. Similar to the test we performed in Section 3.2, we test whether there is a sign change in the partial derivatives f_x and f_y. If we find a sign change in both partial derivatives we conclude that we detected a bifurcation point. Otherwise, we conclude that we probably found a turning point of the curve.

3.5 Stopping Criteria

We assume that we deal with a closed curve. Therefore, we need to find out if and when we returned to a part of a branch we have traced already. If we do not include this exit test (in Line 20), we will trace the curve over and over again, ending up in an infinite loop.

Because of the nature of a numeric algorithm it is very unlikely to return to the exact same point from where we started. Therefore, we cannot just test for equality. We also do not want to test the proximity

of the new point p_c to every one of the computed values along the curve. That might be possible for the first few values, when the cost is still low, but certainly not for arbitrary curves, where we have to compute many points to reasonably approximate the curves shape. The other problem is again the meaning of "very close". To solve this problem, we maintain a counter that counts the number of times we attempt to draw a pixel that has been drawn already. When the algorithm computes a point on the curve that is mapped to a pixel that was not drawn before, we set the counter to zero. Every time we compute a new point on the curve within a pixel that was previously drawn, we increase a counter by one. If the counter's value exceeds some threshold, the algorithm stops pursuing this path.

TABLE 1. Performance results comparing our algorithm. The empty cells denote cases where Chandler's algorithm could not draw the curve.

Figure	# Function evaluation per pixel		# pixels per predictor	# pixels per predictor
	Our Method	Chandler		
5a	1.89	2.68	6.21	2.55
5b	2.01	2.80	6.79	3.20
5c	3.27	2.92	3.51	2.49
6a	2.88	–	6.66	5.04
6b	4.18	–	4.05	3.67
6c	4.55	–	2.38	2.30

4. Results

We applied our algorithm to many different functions with and without bifurcation points and found that they were drawn correctly in any image resolution of 128^2 to 512^2. For images of low resolution (32^2 and sometimes 64^2) the user may have to explicitly specify a smaller step size. We have chosen three very common functions without a bifurcation point (Figure 5) and three interesting functions with many bifurcation points (Figure 6). Some interesting statistics for each group of functions is summarized in Table 1 and compared to Chandlers algorithm [7]. The test cases where drawn onto a 5122 image. Drawing times, for all examples, was under 0.2 seconds on a Silicon Graphics Crimson (150Mhz R4400). In the table we also provide the average number of pixels per one predictor step (showing the adaptive step size), and the average number of correct steps per predictor.

In general, the number of function evaluations per pixel for low resolutions is expectably high, since the curvature of a curve considered within a pixel is larger than if the same curve is subdivided into smaller parts (i.e. for smaller pixel sizes). Therefore, the step size (in terms of the number of pixels) is smaller and the algorithm must perform more calculations on a per pixel basis. For high resolutions we have much fewer function evaluations per pixel than Chandler's algorithm without loss of quality. This demonstrates the power of the adaptive step size.

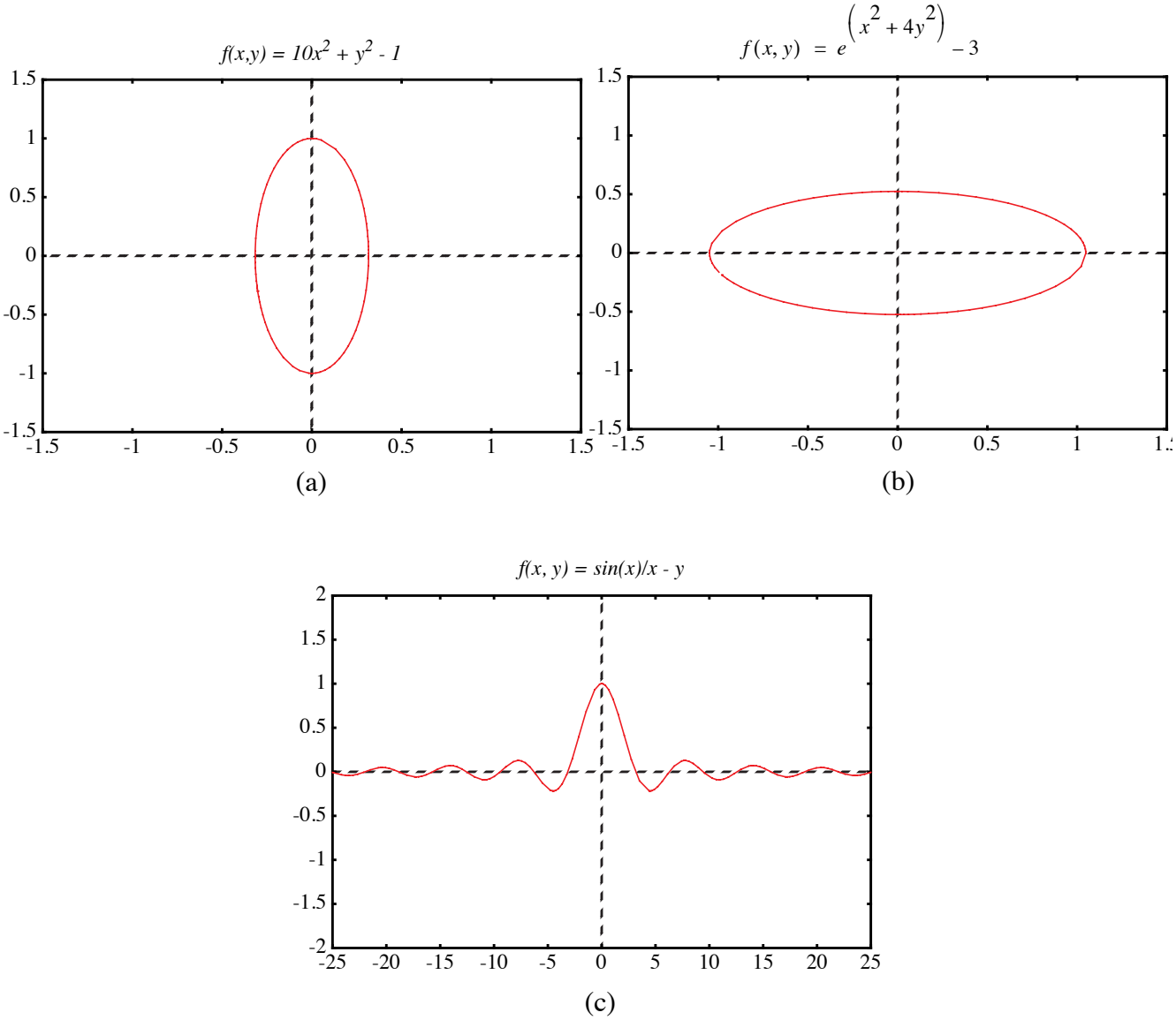

FIGURE 5. Curves without bifurcation points.

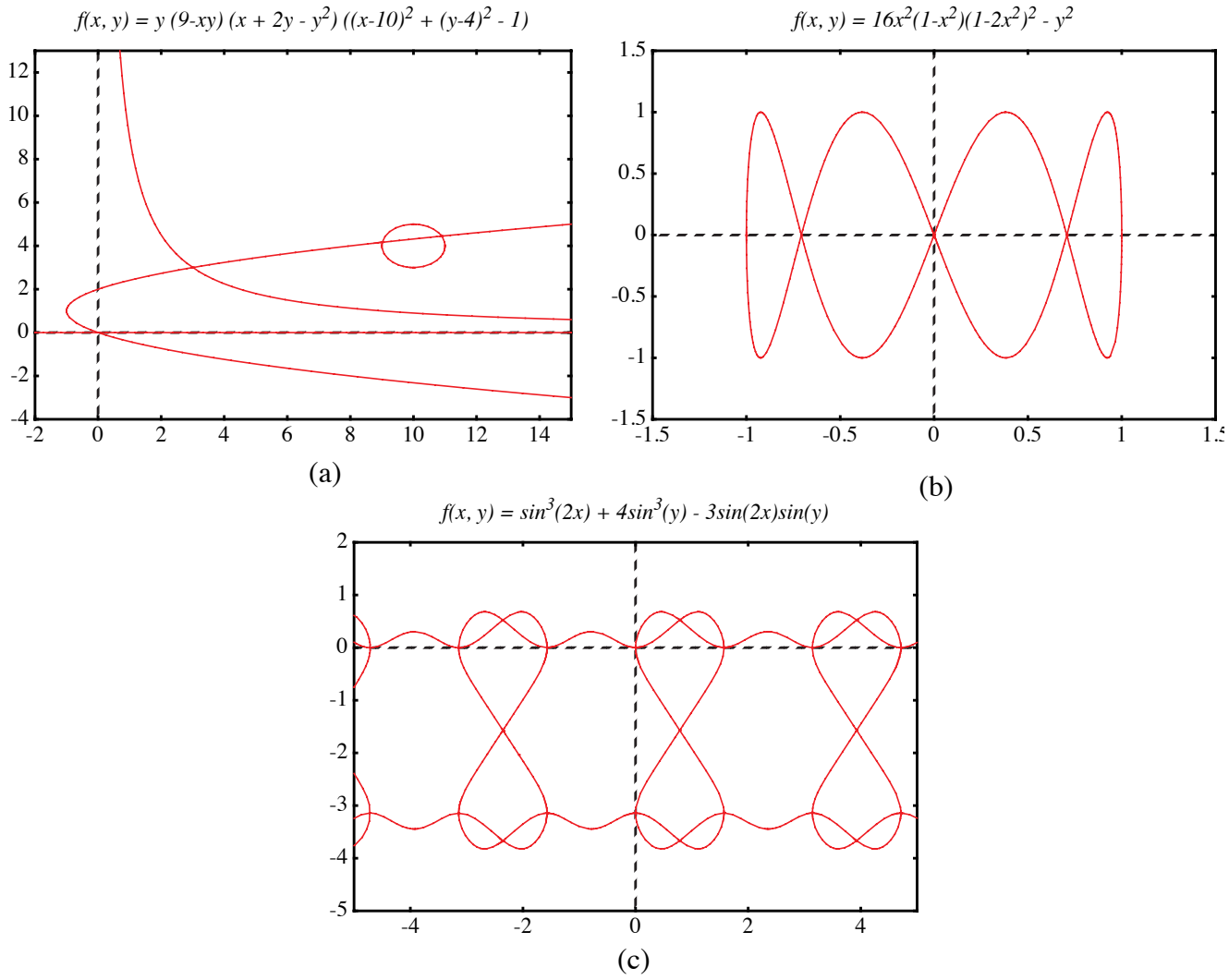

FIGURE 6. Curves with bifurcation points.

5. Conclusions

We presented here a numerical algorithm that is able to draw a large family of implicit functions. Through an adaptive step size design our performance is greatly optimized. We have shown that our approach outperforms previous tracking algorithms.

Our future research includes the improvement of the accuracy of the predictor step. Since we already know the derivatives at the previous point and, most of the time, also have this information available for the point before that, we could use the Hermite interpolation scheme and approximate our curve with a

polynom of third degree. This higher order approximation of our curve should result in fewer corrector steps. Since the higher order polynomial is computed only at the predictor step, saving several corrector steps, we expect a more efficient algorithm. A similar improvement can be made when connecting a new point on the curve with the previous one. Instead of a line through these point, one could use a higher order polynomial. Improvements to the accuracy of the corrector procedure are also possible, for example, by using a better corrector than Newton's. Finally, we plan to explore the extension of our methods to 3D voxelization of implicit surfaces.

6. References

[1] Allgower E. and S. Gnutzmann, "Simplicial Pivoting for Mesh Generation of Implicitly Defined Surfaces", Computer Aided Geometric Design, 8(4), 1991, pp. 30.

[2] Arvo J. and K. Novins, "Iso-Contour Volume Rendering", Proceedings of 1994 Symposium On Volume Visualization.

[3] Baker G. and E. Overman, "The Art of Scientific Computing", Draft Edition, The Ohio State Bookstore.

[4] Blinn J.F., "A Generalization of Algebraic Surface Drawing", ACM Transactions on Graphics, 1(3), 1982, pp. 235-256.

[5] Bloomenthal J., "Polygonization of Implicit Surfaces", Computer Aided Geometric Design, 5(4), 1988, pp. 341-356.

[6] Bronstein I. N. and Semendjajew, K. A. "Taschenbuch der Mathematik", BSB B.G.Teubner Verlagsgesellschaft, Leipzig, 1985.

[7] Chandler R.E., "A Tracking Algorithm for Implicitly Defined Curves", IEEE Computer Graphics and Applications, 8(2), March 1988, pp. 83-89.

[8] Foley J. D., A. van Dam, S. K. Feiner and J. F. Hughes, "Computer Graphics Principles and Practice", Addison-Wesley, 1990.

[9] Jordan B.W., W.J. Lennon, and B.D. Holm, "An Improved Algorithm for the Generation of Nonparametric Curves", IEEE Trans. Computers, Vol. C-22, 1973, pp. 1052-1060.

[10] Hall M. and J.Warren, "Adaptive Polygonization of Implicitly Defined Surfaces", IEEE Computer Graphics and Applications, 10(6), November 1990, pp. 33-42.

[11] Hanrahan P., "Ray Tracing Algebraic Surfaces", Computer Graphics (SIGGRAPH'83 Proceedings), 17(3), 1983, pp. 83-90.

[12] J.C. Hart. Ray Tracing Implicit Surfaces. WSU Technical Report EECS-93-014. Chapter in "Design, Visualization, and Animation of Implicit Surfaces," ACM SIGGRAPH '93 intermediate course notes, J. Bloomenthal and B. Wyvill, eds., Aug. 1993.

[13] Hobby J.D., "Rasterization of Nonparametric Curves", ACM Transactions on Graphics, 9(3), July 1990, pp. 262-277.

[14] Kalra D. and A.H. Barr, "Guaranteed Ray Intersections with Implicit Surfaces", Computer Graphics (SIGGRAPH'89 Proceedings), 23, 1989, pp. 297-306.

[15] Lorensen W. and H. Cline, “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”, Computer Graphics, 21(4), 1987, pp. 163-169.

[16] Sederberg T.W., D.C.Anderson, and R.N.Goldman, “Implicitization, Inversion, and Intersection of Planar Rational Cubic Curves”, Computer Vision, Graphics, and Image Processing, 31(1), 1985, pp. 89-102.

[17] Sederberg T.W. and A.K.Zundel, “Scan Line Display of Algebraic Surfaces”, Computer Graphics (SIGGRAPH’89 Proceedings), 23(4), 1989, pp. 147-156.

Efficient Rasterization of Implicit Functions

Torsten Möller and Roni Yagel

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio

{moeller, yagel}@cis.ohio-state.edu

Abstract

Implicit curves are widely used in computer graphics because of their powerful features for modeling and their ability for general function description. The most popular rasterization techniques for implicit curves are space subdivision and curve tracking. In this paper we are introducing an efficient curve tracking algorithm that is also more robust then existing methods. We employ the Predictor-Corrector Method on the implicit function to get a very accurate curve approximation in a short time. Speedup is achieved by adapting the step size to the curvature. In addition, we provide mechanisms to detect and properly handle bifurcation points, where the curve intersects itself. Finally, the algorithm allows the user to trade-off accuracy for speed and vice a versa. We conclude by providing examples that demonstrate the capabilities of our algorithm.

1. Introduction

Implicit surfaces are surfaces that are defined by implicit functions. An implicit function f is a mapping from an n dimensional space R^n to the space R, described by an expression in which all variables appear on one side of the equation. This is a very general way of describing a mapping. An n dimensional implicit surface S is defined by all the points p in R^n such that the implicit mapping f projects p to 0. More formally, one can describe S by

$$S_f = \{p \mid p \in R^n, f(p) = 0\} . \quad (1)$$

Often times, one is interested in visualizing *isosurfaces* (or *isocontours* for 2D functions) which are functions of the form $f(p) = c$ for some constant c.

Different classes of implicit surfaces are of importance. Implicit surfaces that are defined by a polynomial are called *algebraic surfaces*. Since almost all continuous functions can be approximated by polynomials, algebraic surfaces represent a powerful class. Since polynomials have been studied and understood fairly well, one has a better mathematical handle at algebraic surfaces.

Another special class of implicit surfaces have density functions as their defining functions. Density functions decrease exponentially with increasing distance to a certain center point. The models where these functions find usage usually arise in the study of physical phenomena, e.g. atom models. Blinn [4] was one of the first to try to visualize these surfaces on a computer. They became to be known as Blinn's "*Blobby Model*" or *Metaballs*.

Yet another model which is gaining importance is the *discrete data set*. Such data set is commonly generated by various medical scanners such as MRI and CT. These data sets can also be seen as implicit functions, where $f(p) = v$ and v is the scanned value at the location p. We assume that if p is not on a grid point then some sort of interpolation from neighboring grid points defines v at p.

Rendering implicit surfaces is very difficult and computationaly very expensive. Usually existing algorithms to render implicit surfaces work only for special cases or special sets of functions. In fact, to date there is no such algorithm that can render arbitrary implicit surfaces accurately and in a stable and timely manner.

Algorithms for rendering 3D implicit surfaces can be classified into three groups: representation transforms, ray tracing, and surface tracking. The first approach tries to transform the problem from implicit surfaces to another representation which is easier to display such as polygon meshes [1][5][10][15] or parametric representations [16]. The second approach to rendering of implicit surfaces is to ray-trace them. Efficient ray-object intersection algorithms have been devised for algebraic surfaces [11], metaballs [4], and a class of functions for which Lipschitz constants can be efficiently computed [12][14]. The third approach to rendering implicit functions is to scan convert or discretize the function. Seder-

berg and Zundel [17] introduced such a scan line algorithm for a class of implicit functions that can be represented by Bernstein polynomials.

When it comes to drawing 2D implicit functions (curves) an attractive approach is to follow the path of the curve. The algorithm starts at a seed point on the curve. It then looks at the pixels adjacent to the seed point to see if any of them is the next pixel along the curve. Various algorithms exist which mainly differ in the method they use to find the next pixel along the curve.

One way to find the next pixel is based on *digital difference analysis* (DDA). A lot of work has been done in this field and the most popular algorithm applying this technique are Bresenham's and the midpoint line and circle drawing algorithms [8]. The rate of change in x and y is computed through fast integral updates of dx and dy, which are then used as decision variables to determine the next pixel center. This idea was generalized for algebraic curves by Hobby [13]. One problem of this approach is that the inherent integer arithmetic can only be exploited for a small set of algebraic functions. Furthermore, this algorithm is not able to deal properly with self intersecting curves. Hobby assumes that these intersection points are given by the user.

A different way to continue a curve would be to evaluate the underlying function at the center of the neighboring pixels to find the one with the smallest absolute function value. This idea has been implemented by Jordan et. al. [9] and was improved by Chandler [7]. Since there are only eight neighboring pixels, eight function evaluations are the maximum needed to find the next pixel on the curve. Since we will very likely be traveling in the same direction we have been traveling when we picked the current pixel, Chandler tests first the pixel continuing in the same direction, then its neighbors etc. For most 'nicely' behaving functions two to three function evaluations per pixel will be sufficient. To increase accuracy, Chandler prefers to look for a sign change rather than the smallest absolute value. Therefore, he evaluates the function not at pixel centers but rather halfway in between pixels. Although the algorithm does not depend on the actual description of the function and therefore is applicable to arbitrary functions, it is still not able to deal with self intersecting functions. To deal with such points the algorithm requires user intervention. Alternatively, it includes a brute-force Monte Carlo method to visualize the positive and negative regions of the function. Finally, if the curve enters and leaves the pixel in-

between the same two neighboring sample points, this algorithm will not be able to register a sign change and therefore fail.

For density functions, Arvo et. al. applied a numerical method known as the predictor-corrector method [2]. Their main goal is the rendering of iso-curves of medical data sets. In this paper we introduce an efficient and more robust predictor-corrector method. Our algorithm deals with self intersecting curves properly and increases the efficiency of the algorithm to less than one function evaluation per pixel on the average. Finally, unlike existing curve tracking methods, our algorithm can be applied to a large family of implicit functions including selfintersecting functions with at most two branches at the intersection point.

2. The Predictor-Corrector Method

The predictor-corrector method is a well known numerical algorithm which has its origin in solving ordinary differential equations and bifurcation theory. Arvo and Novins [2] used a two dimensional standard method for extracting isocurves in a 2D medical image. Although the predictor-corrector method offers ways to detect and properly deal with self intersecting curves and curves “very close” to themselves, Arvo and Novins did not explore these capabilities.

The predictor-corrector method is a numerical continuation method (tracking algorithm). The goal is to find a point set which represents a discretized approximation of a curve that is defined by some implicit function as in Equation (1). As the name suggests, the predictor-corrector method consists of two steps. In the predictor step we take a (more or less sophisticated) guess of a point on the curve, denoted by p_p. Depending on how much thought and work we put into predicting the point p_p, it will be closer or farther away from an actual point on the curve. Depending on the assumption we make about the curve and its actual shape, we will not be able to always accurately predict an exact point on the curve. In the corrector step we use our predicted point p_p as a starting point for an iterative method that converges to an actual point on the curve. We denote this new, corrected point, by p_c. We repeat these two steps until we tracked the whole curve. The resulting discrete point set can then be displayed on a screen.

There are many different ways to implement the predictor or corrector steps. We use the standard Euler-Newton method with an adaptive step size and introduce techniques to handle selfintersecting curves.

2.1 The Predictor Step

The focus of the predictor step is to find a point p_p that is very close or even on the curve, so that the costs for the corrector step are reduced or even eliminated. If we already know points on the curve, we can approximate the curve through a polynomial and get a new approximate for this curve through extrapolation or interpolation.

Finding the very first point on the curve is not always easy. One usually has some understanding of the kind of function one tries to discretize and can therefore provide an initial seed value to the algorithm. If this is not possible, one needs to apply a brute force algorithm or some other heuristic to find an initial seed value. The image space is limited (usually a screen in computer graphics). Therefore one rather brute-force method would be to impose a grid onto the screen and evaluate the underlying implicit function at each grid point to choose a closest point (where $|f(p)|$ becomes smallest). Another possibility would be to look for a sign change of the underlying function between neighboring grid points. We employ a faster method that is based on a quadtree subdivision algorithm as explained in Section 3.1.

We now present one way to implement the predictor step. *One-step methods* only employ one point (the point of the previous iteration) to predict a new point. The standard Euler method is a good example of a one-step method. However, one can envision the use of Hermites method which employs more than one point and is therefore called *multi-step method*. Although multi-step methods are usually more expensive, they lead to better, higher order approximations, reducing the number of necessary corrector steps.

Euler's method starts at a point which is known to be on the curve. Because this is a one-step method, each point on the curve will lead to exactly one new point on the curve. Since we execute many predictor-corrector iterations to obtain a point set that approximates our curve, it is just obvious to use the resulting point of the previous iteration as the initial point for the current iteration. We call this initial point $p_i = (x_i, y_i)$. We also want to have some control over the distance between the new point and

the initial point. That is, we want p_p to be located at a certain distance Δs from p_i and not at some unpredictable "end" of the curve. Δs is called the arc length parameter and it represents a parameterization of our curve. The linear approximation of the predicted point results in

$$\begin{aligned} x_p &= x_i + dx\Delta s \\ y_p &= y_i + dy\Delta s \end{aligned}, \tag{2}$$

where $p_p = (x_p, y_p)$ now depends on the arc length and the initial point. Since Δs is the euclidean distance between p_p and p_i, we need to make sure that:

$$dx^2 + dy^2 = 1 \tag{3}$$

Equation (3) defines a unit circle around the origin. Combining it with Equation (2) we are describing a circle of radius Δs and origin p_i. Next, we have to find the intersection points of that circle with our curve. We choose one of these points as our new predicted value. Since the origin of this circle lies already on the curve, we are guaranteed that our circle will intersect our implicit curve, unless the whole curve is contained within the circle. In that particular case, the choice of the distance Δs between neighboring points is not appropriate and needs to be changed. If Δs is in the order of the size of a pixel and the curve is still contained within such a small circle, the discretization of this curve would just be one pixel and we are done.

Assuming a closed curve, we will actually get an even number of intersection points. Choosing our step size (arc length) small enough, we will get exactly two intersection points (see Figure 1). However, in the case of a self intersecting curve or when a curve comes "close" to itself, we will get more than two intersection points, even if Δs is very small. We will explore these special cases in Section 2.2.

The second condition for the new predicted point is, of course, that it should lie on the curve defined by the underlying implicit function *f*. That means we require

$$f(x_p, y_p) = 0\,. \tag{4}$$

Since we are processing an arbitrary functions *f* we are not able to use Equation (4) directly to compute p_p. Instead, we substitute Equation (2) into Equation (4) and through a Taylor series expansion we are

able to compute an approximation to the actual point on the curve. This approximation serves as the predicted value p_p.

Let's have a look at the equations. The substitution results in

$$f(x_i + dx\Delta s, y_i + dy\Delta s) = 0. \tag{5}$$

Now the actual approximation takes place. We expand Equation (5) in a Taylor series and keep only the linear terms. That results in:

$$f(x_i, y_i) + \frac{\partial}{\partial x} f(x_i, y_i) dx\Delta s + \frac{\partial}{\partial y} f(x_i, y_i) dy\Delta s = 0 \tag{6}$$

The initial point was assumed to be on the curve, i.e. $f(x_i, y_i) = 0$. We conclude

$$\frac{\partial}{\partial x} f(x_i, y_i) dx\Delta s + \frac{\partial}{\partial y} f(x_i, y_i) dy\Delta s = 0 \tag{7}$$

Using Equation (3) and Equation (7), we can actually solve for the unknowns (dx, dy). Using a short-cut notation for the partial derivatives of $f(x_i, y_i)$, where $\frac{\partial}{\partial x} f(x_i, y_i)$ is replaced by f_x (and similarly for the partial derivative in y) the result can be written as

$$\begin{aligned} dx &= \delta \frac{f_y}{\sqrt{f_x^2 + f_y^2}} \\ dy &= -\delta \frac{f_x}{\sqrt{f_x^2 + f_y^2}} \end{aligned} \tag{8}$$

which is also proven by Baker and Overman [3]. Here δ stands for the sign plus or minus which represents the direction in which we continue to track the curve. In the case of the seed point we want to make sure, that we trace both directions. However, in the general case we don't want to step back into the direction we were coming from. A simple sign check of the scalar product of the direction, where we were coming from and our new direction, helps us determining the appropriate δ.

Although our new predicted value lies on a circle around the initial point, we only find a point near the curve because of the approximation in Equation (6). If we take a closer look at Equation (7), the one we actually used as an approximation to Equation (5), we realize that (dx, dy) describes the components

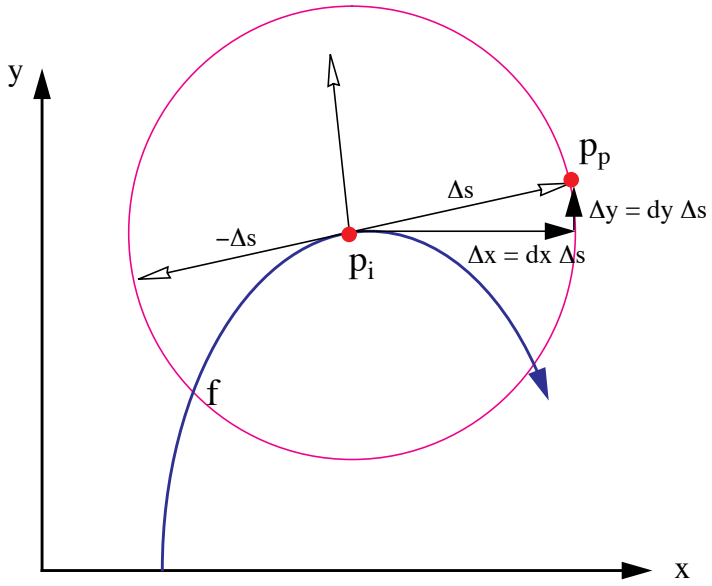

FIGURE 1. The predictor step. Here Δs is the step size we want to step along the curve starting at the initial point p_i. Because Equation (7) is only an approximation, the predicted value will not necessarily lie on the curve described by f.

of the tangent to the curve at the initial point. That leads us to a better geometric understanding of this process, captured in Figure 1.

2.2 Bifurcation Points

The intersection of the circle equation (originating in Equation (3)) with the implicit curve results usually in two points. As pointed out earlier, in case of a selfintersecting curve (Figure 2a) or if it just comes "close" to itself (Figure 2b) we expect at least four intersection points. These special points are also called *bifurcation points*. Looking at our formulas in Equation (7) and Equation (3), we are only able to compute two points, since it is an intersection of a quadratic (Equation (3)) curve with a linear curve (Equation (7)). In a case of a bifurcation point the linear approximation of the Taylor series as in Equation (7) will not be enough. In fact, from the implicit function theorem we can conclude, that the first partial derivatives will be zero for these bifurcation points. Therefore we cannot compute (dx, dy) as derived above. We have to include the second order terms of the Taylor series expansion in Equation (6) to get the appropriate results.

The more accurate approximation to Equation (5), replacing Equation (6), is now

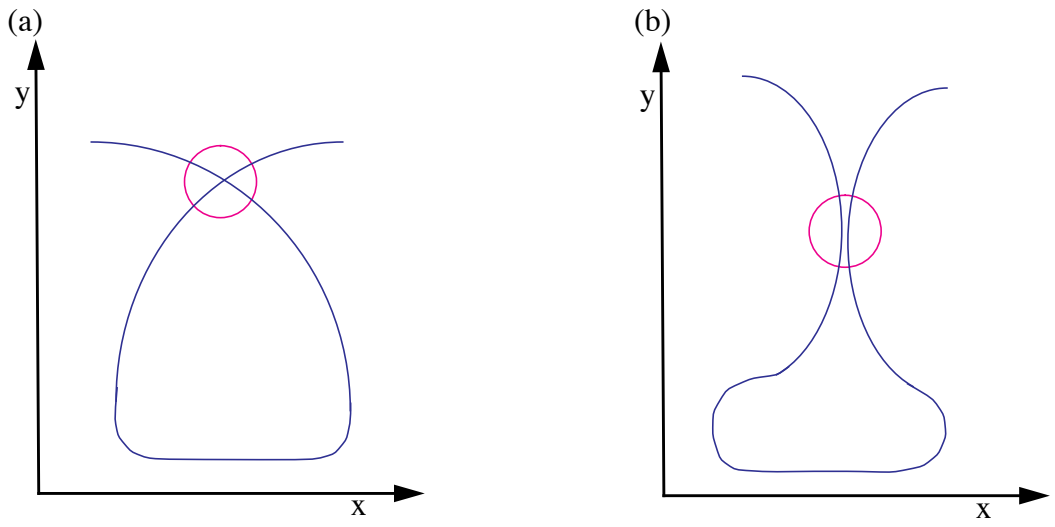

FIGURE 2. Possible cases of bifurcation points where the circle represents the actual step size. (a) The curve intersects itself; (b) The curve comes very “close” to itself.

$$\begin{aligned} &f(x_i, y_i) + f_x(x_i, y_i)dx\Delta s + f_y(x_i, y_i)dy\Delta s + \\ &f_{xx}(x_i, y_i)dx^2\Delta s^2 + 2f_{xy}(x_i, y_i)dxdy\Delta s^2 + f_{yy}(x_i, y_i)dy^2\Delta s^2 = 0 \end{aligned} \tag{9}$$

As already mentioned, the first three terms in the sum vanish, which leads to

$$f_{xx}(x_i, y_i)dx^2\Delta s^2 + 2f_{xy}(x_i, y_i)dxdy\Delta s^2 + f_{yy}(x_i, y_i)dy^2\Delta s^2 = 0 \tag{10}$$

In order to increase the readability of the following formulas, we use the following notation:

$$\begin{aligned} A &= f_{xx}(x_i, y_i) \\ B &= f_{xy}(x_i, y_i) \\ C &= f_{yy}(x_i, y_i) \end{aligned} \tag{11}$$

If we consider Equation (10) as a quadratic equation in the quotient q defined by dx/dy, we get the following solution:

$$q_{1,2} = \frac{dx}{dy} = \frac{-B \pm \sqrt{B^2 - AC}}{A} = \frac{C}{-B \mp \sqrt{B^2 - AC}} \tag{12}$$

To actually compute dx and dy, we substitute this result in Equation (3), which results in

$$\begin{pmatrix} dx \\ dy \end{pmatrix} = \pm \begin{pmatrix} \sqrt{\dfrac{1}{\dfrac{1}{q_{1,2}^2}+1}} \\ \sqrt{\dfrac{1}{q_{1,2}^2+1}} \end{pmatrix} = \pm \begin{pmatrix} \sqrt{\dfrac{C^2 - AC + 2B\,(B \pm \sqrt{B^2 - AC})}{(A-C)^2 + 4B^2}} \\ \sqrt{\dfrac{A^2 - AC + 2B\,(B \mp \sqrt{B^2 - AC})}{(A-C)^2 + 4B^2}} \end{pmatrix} \tag{13}$$

where any arbitrary combination of plus and minus from Equation (12) and Equation (13) results in four different distinct direction vectors (dx, dy). We will store all four directions in a list and follow their branches one at a time. This is explained in greater detail in Section 3.

We discussed the usual case of two intersection points and studied in detail what happens when these formulas fail when we encounter bifurcation points. Unfortunately, it could happen that we do not have four intersection points, but six, eight, or more. That means, that actually more then the two branches, as shown in Figure 1, meet at the same area. In that case, we realize that the second partial derivatives in Equation (9) vanishes also, leaving us with no choice other than including even higher order terms in the approximation of the Taylor series. This implies that we will have to solve higher order polynomials to produce the expected number of solutions.

2.3 The Corrector Step

Although we have tried hard to predict a new point p_p on the curve, it is not guaranteed to fall accurately on the curve. We can only hope to find a point that is "very close" to the curve. The goal of the *corrector step* is to find a "closest" point on the curve, starting at p_p. Among the many methods for finding zero points of functions, the Newton method is very stable and fast, and therefore very common. The Newton method is a numerical iteration method which converges to a zero point p^* of a function f, starting at a point p_0 that should be fairly close to the zero point already. In our case, the starting point for the Newton method will be p_p – the result of the predictor step.

The predictor step produced an initial value $p_0 = p_p = (x_p, y_p) = (x_0, y_0)$ for the iteration method. At each iteration step we take the new and improved value from the previous iteration and try to improve it even further, so that we actually end up on the curve itself. The result of the current iteration can be written as:

$$p_{k+1} = \begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k + \Delta x \\ y_k + \Delta y \end{pmatrix}, \tag{14}$$

where k marks the iteration number.

But how can we determine $(\Delta x, \Delta y)$? Here we exploit the same idea as we did in Euler's Method. Of course, we want that our new point (x_{k+1}, y_{k+1}) lies on the curve. So we add the condition that p_{k+1} is a curve point:

$$f(x_{k+1}, y_{k+1}) = 0. \tag{15}$$

The same ideas and manipulations as used for Equation (4) will lead us towards a solution. Substituting Equation (14) into Equation (15) and using a Taylor series expansion of f will result in:

$$f_x(x_k, y_k)\Delta x + f_y(x_k, y_k)\Delta y = -f(x_k, y_k). \tag{16}$$

That is, in fact, the formula for Newton's method for a two dimensional function. Since this is one equation with two unknowns, we have the freedom and duty to specify another condition, we want our iterate to satisfy. First of all, we want to make sure that our iteration process converges. Other than that, it would be nice if we could guarantee some sort of constant discretization of our curve. That is, we do not want the distance between samples to vary too much. Both of these ideas can be realized if we restrict our convergence to move in a right angle away from our previous iterate. To be more specific, the triangle formed by the points p_i, p_k and p_{k+1} should form a right angled triangle. Here, p_i is the initial point on the curve – the one we used in the predictor step to compute the point p_p. p_k and p_{k+1} are the previous and current iterates. In Figure 1 it was shown that this condition on our new iterate results in a somewhat circular movement around the initial point p_i. Because of this circular movement, we stay in a certain proximity to p_i, which increases the chances of convergence.

More formally, we want to add the condition that the vector $\overrightarrow{p_k p_i}$ is perpendicular to $\overrightarrow{p_k p_{k+1}}$. That is expressed in the following manner:

$$(p_i - p_k) \bullet (p_{k+1} - p_k) = \begin{bmatrix} x_i - x_k \\ y_i - y_k \end{bmatrix}^T \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \Delta x\,(x_i - x_k) + \Delta y\,(y_i - y_k) = 0 \tag{17}$$

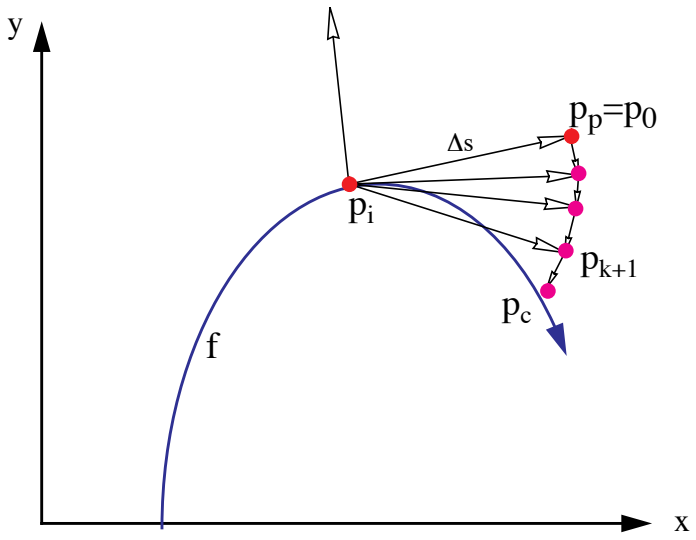

FIGURE 3. The corrector step. The Newton iterations and a forced circular movement of the iterates p_{k+1} converge towards an actual point on the curve p_c.

Equation (16) together with Equation (17) uniquely describe a new, corrected iterate p_{k+1}, which concludes the Corrector step. A geometric interpretation of these steps is illustrated in Figure 3.

3. The Algorithm

The pseudo code for our numerical algorithm is given in Figure 4. The algorithm starts by looking for a seed point (using a space subdivision method, as described in Section 3.1). The initial point will be most likely on some arbitrary part of the curve. Therefore, we will usually have two possible directions we can follow. We record both in a list for later retrieval (Line 3). At a bifurcation point a pixel will have four branches leaving it. In this case we will also record all possible branch directions in this list (Line 18). After retrieving a starting point for a new branch (Line 5) we trace the branch until we find another bifurcation point or until we reach the limits of our display device. In the case of a closed curve we also need to check when we get back to an already traced branch of the curve. A simple way to do this is explained in Section 3.5.

```
1. find an initial point p_i on the surface;
2. determine initial step size Δs;
3. record (p, Δs, bifurlist);                    /* for both initial direction */
4. repeat
5.     get p_i, Δs from bifurlist
6.     repeat                                    /* continue current branch */
7.         repeat                                /* find a reliable new point on the curve */
8.             p_p := Predictor(p_i)
9.             p_c := Corrector(p_p)
10.            C := approx_curvature(p_c, p_p)
11.            Δs := Δs/2
12.        until (p_c converges AND 1/(2C) < Δs) OR bifur_test(p_c)
13.        Δs := Δs*2;
14.        if 1/(2C) >Δs then
15.            increase Δs;
16.        draw_line(p_c, p_i);
17.        if bifur_test(p_c) then
18.            record(p_c, Δs, bifurlist)
19.        p_i := p_c;
20.    until (p_c is in the covered region or outside the screen OR bifur_test(p_c));
21. until bifurlist is empty
```

FIGURE 4. pseudo code for the curve tracking algorithm.

For each new predictor point we need to make sure that the corrector didn't fail (Line 7 through Line 12). Once we have gained confidence about our corrected point, we check whether we can adjust the step size (Line 15). We connect the previous point to the current point by a simple line (Line 16). If we found a new bifurcation point, we need to record it (Line 18) and quit this loop. In any other case we repeat this procedure and find another point on the same branch.

We now describe in more detail the implementation of some of the more involved segments of the algorithm.

3.1 Finding a Starting Point

If a seed point is not provided by the user, we employ a space subdivision approach. We divide the image region into four quadrants and compute the function at each corner of the quadrants. If the sign of the function is the same at all corners, we will apply the same subdivision to all four quadrants in a breath-first scheme. If we do find a sign change, we continue to subdivide this specific quadrant in a depth-first scheme up to the pixel- or even subpixel level, depending on how accurate our initial point should be.

3.2 Implementing the Corrector Step

Since the predicted point is only an approximation of a curve point, the corrector step is employed to converge to an actual point on the curve. Because of the approximations in our formulas, the Newton corrector step (explained in Section 2.3) usually needs to be repeated a few times. That is done within the Corrector procedure in Line 9. But how many times should we repeat the Newton step? Of course, we have to fight round off errors caused by the discrete computer arithmetic as well. The quality of our discretized curve greatly depends on our stopping criteria, as well as the successful completion of our algorithm. Therefore, we need to know when $f(p_{k+1})$ gets close enough to zero. The problem is that *close* means very different things for different functions. If we had more knowledge about the function, we could employ an absolute error test, like $f(p_{k+1}) < \varepsilon$. For a general algorithm, we will have no idea how small ε needs to be to assure spatial proximity. This notion of *being close* to the curve might also change at different ranges for the curve.

The method we chose adapts the value of ε and utilizes the fact that we draw to a finite grid. Every once in a while (e.g. every tenth time we call the Corrector procedure) we check the quality of the corrected point p_c. For this quality check we define a rectangular region centered at p_c and check for a sign change of the function f at the corner points. The size of this rectangular region can be a parameter L specified by the user or simply the current grid size. If there is a sign change in the corner points, we know that our curve will be at most $L/\sqrt{2}$ away. Therefore, we could relax our stopping criteria and set ε to $1.1|f(p_{k+1})|$. On the other hand, if we find that our corrected value isn't close enough to our curve, we should strengthen our stopping condition and set ε to $10^{-2}|f(p_{k+1})|$ and redo the corrector

step. This finally results in a relative (adaptive) error test, which is certainly superior than an absolute (static) error test.

3.3 Determining The Step Size Δs

The other crucial and very sensitive parameter of this algorithm is the step size Δs . One big advantage of our algorithm is the freedom to choose the step size. When the curve shows a small curvature we will choose a fairly large Δs , allowing the algorithm to step quickly through this “almost linear” part of the curve. On the other hand, when the curve changes rather quickly, i.e. the curvature of the function is high, we have to decrease Δs appropriately. This adaptive mechanism gives us the ability to capture the nature of our curve in greater detail when necessary.

The mathematical definition of the curvature is

$$C(p) = \lim_{q \to p} \frac{\alpha(p) - \alpha(q)}{\widehat{pq}}, \tag{18}$$

where $\alpha(p)$ denotes the angel of the tangent with the x-axis and $\widehat{pq}$ denotes the arc length between the two points. For implicit functions, C can be computed as

$$C = \frac{-f_y^2 f_{xx} + 2 f_x f_y f_{xy} - f_x^2 f_{yy}}{\left(f_x^2 + f_y^2\right)^{3/2}} \tag{19}$$

(See [6]). We compute first derivatives anyway for the Newton corrector, but would like to avoid computing second derivatives. Therefore, we use an approximation to the curvature that is based on Equation (18), where we use the current and previous point on the curve. Since we have the first derivatives already computed we can precisely compute the tangent angles. The arc length between the two points is simply approximated with the current step size. To compute the initial step size we could either use Equation (19) or just use the grid size. Choosing the initial step size for a bifurcation point can be a problem. If we are stepping too far, we might ‘loose track’ of the current branch. If we are stepping too close, the corrector might converge to a different, neighboring branch. Therefore we left it up to the user to choose this initial step size.

Another indicator of the quality of our step size would be the number of corrector steps we had to take before we converged to a curve point. In our predictor step we are basically following the tangent of the curve. If the curve is fairly linear the predicted point p_p will be almost on the curve, resulting in just very few corrector steps. If the curve changes its direction rather fast, p_p will not be the best guess and we expect more correction steps. Therefore, if we use at most one Newton step to find a point on the curve, we simply double the step size.

3.4 Detecting Bifurcation Points

Bifurcation points are points where the curve intersects itself. At these points the first derivatives of the function evaluate to zero. Of course, it will be very unlikely to find the mathematically exact bifurcation point (in case one exists), because of the numerical nature of the algorithm. Therefore, we need to keep watching the first derivatives of a function for regions where they are *close* to zero. We have seen already how difficult such a test can be (see Section 3.2). Therefore, whenever we find a sign change in one of the partial derivatives (from the previous to the current point) we have good reason to suspect that there might be a bifurcation point close by. As a result, we decrease our step size and repeat the last predictor/corrector step until the step size is smaller than a given size, which can be defined by the user. If there is still reason to believe that there might be a bifurcation point, we create a rectangular region around the current point on the curve. Similar to the test we performed in Section 3.2, we test whether there is a sign change in the partial derivatives f_x and f_y. If we find a sign change in both partial derivatives we conclude that we detected a bifurcation point. Otherwise, we conclude that we probably found a turning point of the curve.

3.5 Stopping Criteria

We assume that we deal with a closed curve. Therefore, we need to find out if and when we returned to a part of a branch we have traced already. If we do not include this exit test (in Line 20), we will trace the curve over and over again, ending up in an infinite loop.

Because of the nature of a numeric algorithm it is very unlikely to return to the exact same point from where we started. Therefore, we cannot just test for equality. We also do not want to test the proximity

of the new point p_c to every one of the computed values along the curve. That might be possible for the first few values, when the cost is still low, but certainly not for arbitrary curves, where we have to compute many points to reasonably approximate the curves shape. The other problem is again the meaning of "very close". To solve this problem, we maintain a counter that counts the number of times we attempt to draw a pixel that has been drawn already. When the algorithm computes a point on the curve that is mapped to a pixel that was not drawn before, we set the counter to zero. Every time we compute a new point on the curve within a pixel that was previously drawn, we increase a counter by one. If the counter's value exceeds some threshold, the algorithm stops pursuing this path.

TABLE 1. Performance results comparing our algorithm. The empty cells denote cases where Chandler's algorithm could not draw the curve.

Figure	# Function evaluation per pixel		# pixels per predictor	# pixels per predictor
	Our Method	Chandler		
5a	1.89	2.68	6.21	2.55
5b	2.01	2.80	6.79	3.20
5c	3.27	2.92	3.51	2.49
6a	2.88	–	6.66	5.04
6b	4.18	–	4.05	3.67
6c	4.55	–	2.38	2.30

4. Results

We applied our algorithm to many different functions with and without bifurcation points and found that they were drawn correctly in any image resolution of 128^2 to 512^2. For images of low resolution (32^2 and sometimes 64^2) the user may have to explicitly specify a smaller step size. We have chosen three very common functions without a bifurcation point (Figure 5) and three interesting functions with many bifurcation points (Figure 6). Some interesting statistics for each group of functions is summarized in Table 1 and compared to Chandlers algorithm [7]. The test cases where drawn onto a 5122 image. Drawing times, for all examples, was under 0.2 seconds on a Silicon Graphics Crimson (150Mhz R4400). In the table we also provide the average number of pixels per one predictor step (showing the adaptive step size), and the average number of correct steps per predictor.

In general, the number of function evaluations per pixel for low resolutions is expectably high, since the curvature of a curve considered within a pixel is larger than if the same curve is subdivided into smaller parts (i.e. for smaller pixel sizes). Therefore, the step size (in terms of the number of pixels) is smaller and the algorithm must perform more calculations on a per pixel basis. For high resolutions we have much fewer function evaluations per pixel than Chandler's algorithm without loss of quality. This demonstrates the power of the adaptive step size.

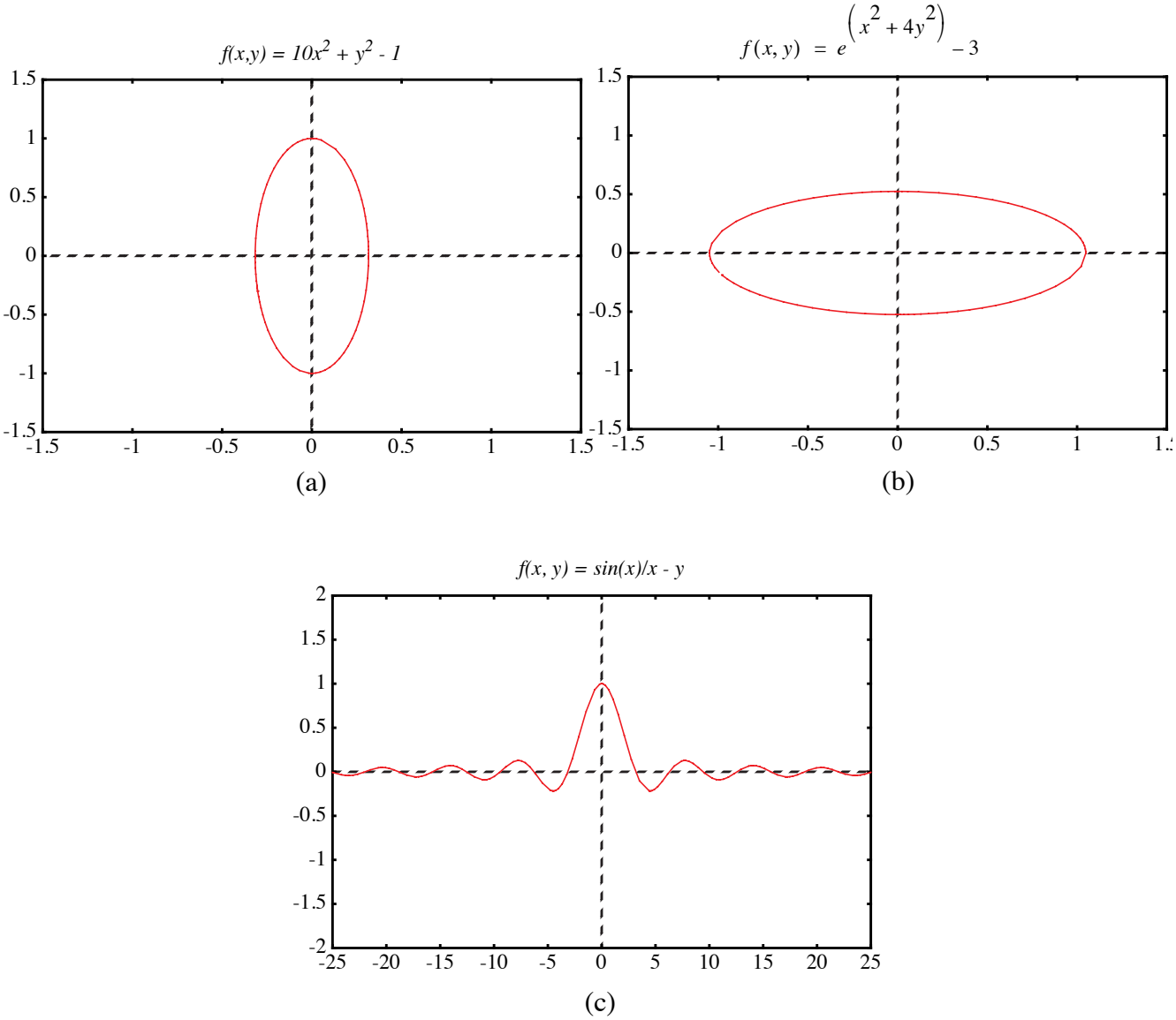

FIGURE 5. Curves without bifurcation points.

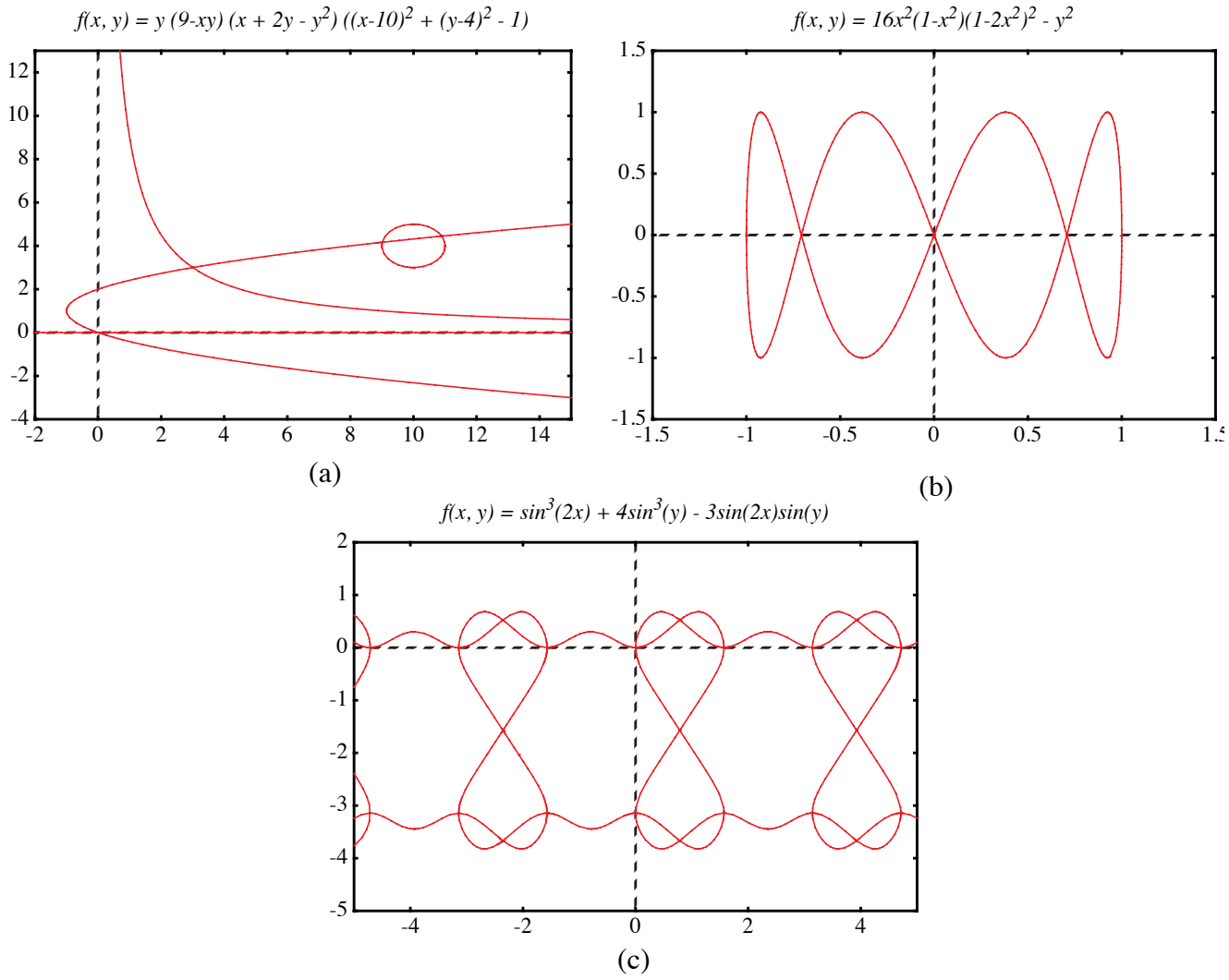

FIGURE 6. Curves with bifurcation points.

5. Conclusions

We presented here a numerical algorithm that is able to draw a large family of implicit functions. Through an adaptive step size design our performance is greatly optimized. We have shown that our approach outperforms previous tracking algorithms.

Our future research includes the improvement of the accuracy of the predictor step. Since we already know the derivatives at the previous point and, most of the time, also have this information available for the point before that, we could use the Hermite interpolation scheme and approximate our curve with a

polynom of third degree. This higher order approximation of our curve should result in fewer corrector steps. Since the higher order polynomial is computed only at the predictor step, saving several corrector steps, we expect a more efficient algorithm. A similar improvement can be made when connecting a new point on the curve with the previous one. Instead of a line through these point, one could use a higher order polynomial. Improvements to the accuracy of the corrector procedure are also possible, for example, by using a better corrector than Newton's. Finally, we plan to explore the extension of our methods to 3D voxelization of implicit surfaces.

6. References

[1] Allgower E. and S. Gnutzmann, "Simplicial Pivoting for Mesh Generation of Implicitly Defined Surfaces", Computer Aided Geometric Design, 8(4), 1991, pp. 30.

[2] Arvo J. and K. Novins, "Iso-Contour Volume Rendering", Proceedings of 1994 Symposium On Volume Visualization.

[3] Baker G. and E. Overman, "The Art of Scientific Computing", Draft Edition, The Ohio State Bookstore.

[4] Blinn J.F., "A Generalization of Algebraic Surface Drawing", ACM Transactions on Graphics, 1(3), 1982, pp. 235-256.

[5] Bloomenthal J., "Polygonization of Implicit Surfaces", Computer Aided Geometric Design, 5(4), 1988, pp. 341-356.

[6] Bronstein I. N. and Semendjajew, K. A. "Taschenbuch der Mathematik", BSB B.G.Teubner Verlagsgesellschaft, Leipzig, 1985.

[7] Chandler R.E., "A Tracking Algorithm for Implicitly Defined Curves", IEEE Computer Graphics and Applications, 8(2), March 1988, pp. 83-89.

[8] Foley J. D., A. van Dam, S. K. Feiner and J. F. Hughes, "Computer Graphics Principles and Practice", Addison-Wesley, 1990.

[9] Jordan B.W., W.J. Lennon, and B.D. Holm, "An Improved Algorithm for the Generation of Nonparametric Curves", IEEE Trans. Computers, Vol. C-22, 1973, pp. 1052-1060.

[10] Hall M. and J.Warren, "Adaptive Polygonization of Implicitly Defined Surfaces", IEEE Computer Graphics and Applications, 10(6), November 1990, pp. 33-42.

[11] Hanrahan P., "Ray Tracing Algebraic Surfaces", Computer Graphics (SIGGRAPH'83 Proceedings), 17(3), 1983, pp. 83-90.

[12] J.C. Hart. Ray Tracing Implicit Surfaces. WSU Technical Report EECS-93-014. Chapter in "Design, Visualization, and Animation of Implicit Surfaces," ACM SIGGRAPH '93 intermediate course notes, J. Bloomenthal and B. Wyvill, eds., Aug. 1993.

[13] Hobby J.D., "Rasterization of Nonparametric Curves", ACM Transactions on Graphics, 9(3), July 1990, pp. 262-277.

[14] Kalra D. and A.H. Barr, "Guaranteed Ray Intersections with Implicit Surfaces", Computer Graphics (SIGGRAPH'89 Proceedings), 23, 1989, pp. 297-306.

[15] Lorensen W. and H. Cline, “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”, Computer Graphics, 21(4), 1987, pp. 163-169.

[16] Sederberg T.W., D.C.Anderson, and R.N.Goldman, “Implicitization, Inversion, and Intersection of Planar Rational Cubic Curves”, Computer Vision, Graphics, and Image Processing, 31(1), 1985, pp. 89-102.

[17] Sederberg T.W. and A.K.Zundel, “Scan Line Display of Algebraic Surfaces”, Computer Graphics (SIGGRAPH’89 Proceedings), 23(4), 1989, pp. 147-156.

Efficient Rasterization of Implicit Functions

Torsten Möller and Roni Yagel

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio

{moeller, yagel}@cis.ohio-state.edu

Abstract

Implicit curves are widely used in computer graphics because of their powerful features for modeling and their ability for general function description. The most popular rasterization techniques for implicit curves are space subdivision and curve tracking. In this paper we are introducing an efficient curve tracking algorithm that is also more robust then existing methods. We employ the Predictor-Corrector Method on the implicit function to get a very accurate curve approximation in a short time. Speedup is achieved by adapting the step size to the curvature. In addition, we provide mechanisms to detect and properly handle bifurcation points, where the curve intersects itself. Finally, the algorithm allows the user to trade-off accuracy for speed and vice a versa. We conclude by providing examples that demonstrate the capabilities of our algorithm.

1. Introduction

Implicit surfaces are surfaces that are defined by implicit functions. An implicit function f is a mapping from an n dimensional space R^n to the space R, described by an expression in which all variables appear on one side of the equation. This is a very general way of describing a mapping. An n dimensional implicit surface S is defined by all the points p in R^n such that the implicit mapping f projects p to 0. More formally, one can describe S by

$$S_f = \{p \mid p \in R^n, f(p) = 0\} . \quad (1)$$

Often times, one is interested in visualizing *isosurfaces* (or *isocontours* for 2D functions) which are functions of the form $f(p) = c$ for some constant c.

Different classes of implicit surfaces are of importance. Implicit surfaces that are defined by a polynomial are called *algebraic surfaces*. Since almost all continuous functions can be approximated by polynomials, algebraic surfaces represent a powerful class. Since polynomials have been studied and understood fairly well, one has a better mathematical handle at algebraic surfaces.

Another special class of implicit surfaces have density functions as their defining functions. Density functions decrease exponentially with increasing distance to a certain center point. The models where these functions find usage usually arise in the study of physical phenomena, e.g. atom models. Blinn [4] was one of the first to try to visualize these surfaces on a computer. They became to be known as Blinn's "*Blobby Model*" or *Metaballs*.

Yet another model which is gaining importance is the *discrete data set*. Such data set is commonly generated by various medical scanners such as MRI and CT. These data sets can also be seen as implicit functions, where $f(p) = v$ and v is the scanned value at the location p. We assume that if p is not on a grid point then some sort of interpolation from neighboring grid points defines v at p.

Rendering implicit surfaces is very difficult and computationaly very expensive. Usually existing algorithms to render implicit surfaces work only for special cases or special sets of functions. In fact, to date there is no such algorithm that can render arbitrary implicit surfaces accurately and in a stable and timely manner.

Algorithms for rendering 3D implicit surfaces can be classified into three groups: representation transforms, ray tracing, and surface tracking. The first approach tries to transform the problem from implicit surfaces to another representation which is easier to display such as polygon meshes [1][5][10][15] or parametric representations [16]. The second approach to rendering of implicit surfaces is to ray-trace them. Efficient ray-object intersection algorithms have been devised for algebraic surfaces [11], metaballs [4], and a class of functions for which Lipschitz constants can be efficiently computed [12][14]. The third approach to rendering implicit functions is to scan convert or discretize the function. Seder-

berg and Zundel [17] introduced such a scan line algorithm for a class of implicit functions that can be represented by Bernstein polynomials.

When it comes to drawing 2D implicit functions (curves) an attractive approach is to follow the path of the curve. The algorithm starts at a seed point on the curve. It then looks at the pixels adjacent to the seed point to see if any of them is the next pixel along the curve. Various algorithms exist which mainly differ in the method they use to find the next pixel along the curve.

One way to find the next pixel is based on *digital difference analysis* (DDA). A lot of work has been done in this field and the most popular algorithm applying this technique are Bresenham's and the midpoint line and circle drawing algorithms [8]. The rate of change in x and y is computed through fast integral updates of dx and dy, which are then used as decision variables to determine the next pixel center. This idea was generalized for algebraic curves by Hobby [13]. One problem of this approach is that the inherent integer arithmetic can only be exploited for a small set of algebraic functions. Furthermore, this algorithm is not able to deal properly with self intersecting curves. Hobby assumes that these intersection points are given by the user.

A different way to continue a curve would be to evaluate the underlying function at the center of the neighboring pixels to find the one with the smallest absolute function value. This idea has been implemented by Jordan et. al. [9] and was improved by Chandler [7]. Since there are only eight neighboring pixels, eight function evaluations are the maximum needed to find the next pixel on the curve. Since we will very likely be traveling in the same direction we have been traveling when we picked the current pixel, Chandler tests first the pixel continuing in the same direction, then its neighbors etc. For most 'nicely' behaving functions two to three function evaluations per pixel will be sufficient. To increase accuracy, Chandler prefers to look for a sign change rather than the smallest absolute value. Therefore, he evaluates the function not at pixel centers but rather halfway in between pixels. Although the algorithm does not depend on the actual description of the function and therefore is applicable to arbitrary functions, it is still not able to deal with self intersecting functions. To deal with such points the algorithm requires user intervention. Alternatively, it includes a brute-force Monte Carlo method to visualize the positive and negative regions of the function. Finally, if the curve enters and leaves the pixel in-

between the same two neighboring sample points, this algorithm will not be able to register a sign change and therefore fail.

For density functions, Arvo et. al. applied a numerical method known as the predictor-corrector method [2]. Their main goal is the rendering of iso-curves of medical data sets. In this paper we introduce an efficient and more robust predictor-corrector method. Our algorithm deals with self intersecting curves properly and increases the efficiency of the algorithm to less than one function evaluation per pixel on the average. Finally, unlike existing curve tracking methods, our algorithm can be applied to a large family of implicit functions including selfintersecting functions with at most two branches at the intersection point.

2. The Predictor-Corrector Method

The predictor-corrector method is a well known numerical algorithm which has its origin in solving ordinary differential equations and bifurcation theory. Arvo and Novins [2] used a two dimensional standard method for extracting isocurves in a 2D medical image. Although the predictor-corrector method offers ways to detect and properly deal with self intersecting curves and curves “very close” to themselves, Arvo and Novins did not explore these capabilities.

The predictor-corrector method is a numerical continuation method (tracking algorithm). The goal is to find a point set which represents a discretized approximation of a curve that is defined by some implicit function as in Equation (1). As the name suggests, the predictor-corrector method consists of two steps. In the predictor step we take a (more or less sophisticated) guess of a point on the curve, denoted by p_p. Depending on how much thought and work we put into predicting the point p_p, it will be closer or farther away from an actual point on the curve. Depending on the assumption we make about the curve and its actual shape, we will not be able to always accurately predict an exact point on the curve. In the corrector step we use our predicted point p_p as a starting point for an iterative method that converges to an actual point on the curve. We denote this new, corrected point, by p_c. We repeat these two steps until we tracked the whole curve. The resulting discrete point set can then be displayed on a screen.

There are many different ways to implement the predictor or corrector steps. We use the standard Euler-Newton method with an adaptive step size and introduce techniques to handle selfintersecting curves.

2.1 The Predictor Step

The focus of the predictor step is to find a point p_p that is very close or even on the curve, so that the costs for the corrector step are reduced or even eliminated. If we already know points on the curve, we can approximate the curve through a polynomial and get a new approximate for this curve through extrapolation or interpolation.

Finding the very first point on the curve is not always easy. One usually has some understanding of the kind of function one tries to discretize and can therefore provide an initial seed value to the algorithm. If this is not possible, one needs to apply a brute force algorithm or some other heuristic to find an initial seed value. The image space is limited (usually a screen in computer graphics). Therefore one rather brute-force method would be to impose a grid onto the screen and evaluate the underlying implicit function at each grid point to choose a closest point (where $|f(p)|$ becomes smallest). Another possibility would be to look for a sign change of the underlying function between neighboring grid points. We employ a faster method that is based on a quadtree subdivision algorithm as explained in Section 3.1.

We now present one way to implement the predictor step. *One-step methods* only employ one point (the point of the previous iteration) to predict a new point. The standard Euler method is a good example of a one-step method. However, one can envision the use of Hermites method which employs more than one point and is therefore called *multi-step method*. Although multi-step methods are usually more expensive, they lead to better, higher order approximations, reducing the number of necessary corrector steps.

Euler's method starts at a point which is known to be on the curve. Because this is a one-step method, each point on the curve will lead to exactly one new point on the curve. Since we execute many predictor-corrector iterations to obtain a point set that approximates our curve, it is just obvious to use the resulting point of the previous iteration as the initial point for the current iteration. We call this initial point $p_i = (x_i, y_i)$. We also want to have some control over the distance between the new point and

the initial point. That is, we want p_p to be located at a certain distance Δs from p_i and not at some unpredictable "end" of the curve. Δs is called the arc length parameter and it represents a parameterization of our curve. The linear approximation of the predicted point results in

$$\begin{aligned} x_p &= x_i + dx\Delta s \\ y_p &= y_i + dy\Delta s \end{aligned}, \tag{2}$$

where $p_p = (x_p, y_p)$ now depends on the arc length and the initial point. Since Δs is the euclidean distance between p_p and p_i, we need to make sure that:

$$dx^2 + dy^2 = 1 \tag{3}$$

Equation (3) defines a unit circle around the origin. Combining it with Equation (2) we are describing a circle of radius Δs and origin p_i. Next, we have to find the intersection points of that circle with our curve. We choose one of these points as our new predicted value. Since the origin of this circle lies already on the curve, we are guaranteed that our circle will intersect our implicit curve, unless the whole curve is contained within the circle. In that particular case, the choice of the distance Δs between neighboring points is not appropriate and needs to be changed. If Δs is in the order of the size of a pixel and the curve is still contained within such a small circle, the discretization of this curve would just be one pixel and we are done.

Assuming a closed curve, we will actually get an even number of intersection points. Choosing our step size (arc length) small enough, we will get exactly two intersection points (see Figure 1). However, in the case of a self intersecting curve or when a curve comes "close" to itself, we will get more than two intersection points, even if Δs is very small. We will explore these special cases in Section 2.2.

The second condition for the new predicted point is, of course, that it should lie on the curve defined by the underlying implicit function *f*. That means we require

$$f(x_p, y_p) = 0\,. \tag{4}$$

Since we are processing an arbitrary functions *f* we are not able to use Equation (4) directly to compute p_p. Instead, we substitute Equation (2) into Equation (4) and through a Taylor series expansion we are

able to compute an approximation to the actual point on the curve. This approximation serves as the predicted value p_p.

Let's have a look at the equations. The substitution results in

$$f(x_i + dx\Delta s, y_i + dy\Delta s) = 0. \tag{5}$$

Now the actual approximation takes place. We expand Equation (5) in a Taylor series and keep only the linear terms. That results in:

$$f(x_i, y_i) + \frac{\partial}{\partial x} f(x_i, y_i) dx\Delta s + \frac{\partial}{\partial y} f(x_i, y_i) dy\Delta s = 0 \tag{6}$$

The initial point was assumed to be on the curve, i.e. $f(x_i, y_i) = 0$. We conclude

$$\frac{\partial}{\partial x} f(x_i, y_i) dx\Delta s + \frac{\partial}{\partial y} f(x_i, y_i) dy\Delta s = 0 \tag{7}$$

Using Equation (3) and Equation (7), we can actually solve for the unknowns (dx, dy). Using a short-cut notation for the partial derivatives of $f(x_i, y_i)$, where $\frac{\partial}{\partial x} f(x_i, y_i)$ is replaced by f_x (and similarly for the partial derivative in y) the result can be written as

$$\begin{aligned} dx &= \delta \frac{f_y}{\sqrt{f_x^2 + f_y^2}} \\ dy &= -\delta \frac{f_x}{\sqrt{f_x^2 + f_y^2}} \end{aligned} \tag{8}$$

which is also proven by Baker and Overman [3]. Here δ stands for the sign plus or minus which represents the direction in which we continue to track the curve. In the case of the seed point we want to make sure, that we trace both directions. However, in the general case we don't want to step back into the direction we were coming from. A simple sign check of the scalar product of the direction, where we were coming from and our new direction, helps us determining the appropriate δ.

Although our new predicted value lies on a circle around the initial point, we only find a point near the curve because of the approximation in Equation (6). If we take a closer look at Equation (7), the one we actually used as an approximation to Equation (5), we realize that (dx, dy) describes the components

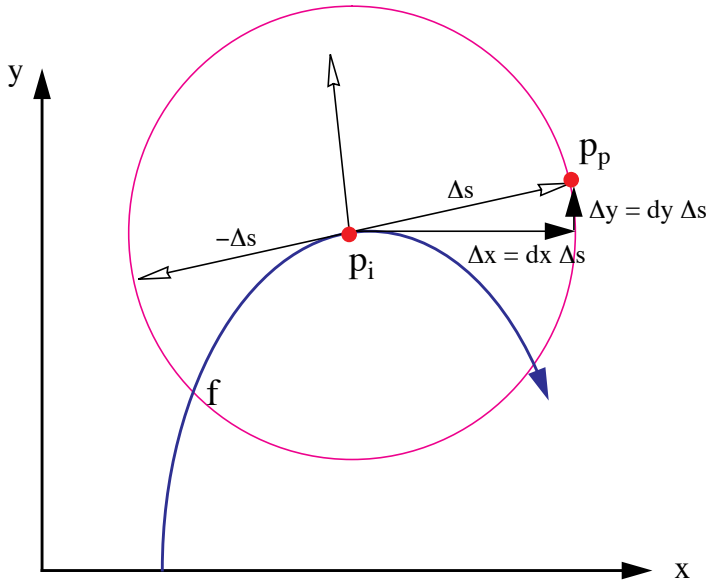

FIGURE 1. The predictor step. Here Δs is the step size we want to step along the curve starting at the initial point p_i. Because Equation (7) is only an approximation, the predicted value will not necessarily lie on the curve described by f.

of the tangent to the curve at the initial point. That leads us to a better geometric understanding of this process, captured in Figure 1.

2.2 Bifurcation Points

The intersection of the circle equation (originating in Equation (3)) with the implicit curve results usually in two points. As pointed out earlier, in case of a selfintersecting curve (Figure 2a) or if it just comes "close" to itself (Figure 2b) we expect at least four intersection points. These special points are also called *bifurcation points*. Looking at our formulas in Equation (7) and Equation (3), we are only able to compute two points, since it is an intersection of a quadratic (Equation (3)) curve with a linear curve (Equation (7)). In a case of a bifurcation point the linear approximation of the Taylor series as in Equation (7) will not be enough. In fact, from the implicit function theorem we can conclude, that the first partial derivatives will be zero for these bifurcation points. Therefore we cannot compute (dx, dy) as derived above. We have to include the second order terms of the Taylor series expansion in Equation (6) to get the appropriate results.

The more accurate approximation to Equation (5), replacing Equation (6), is now

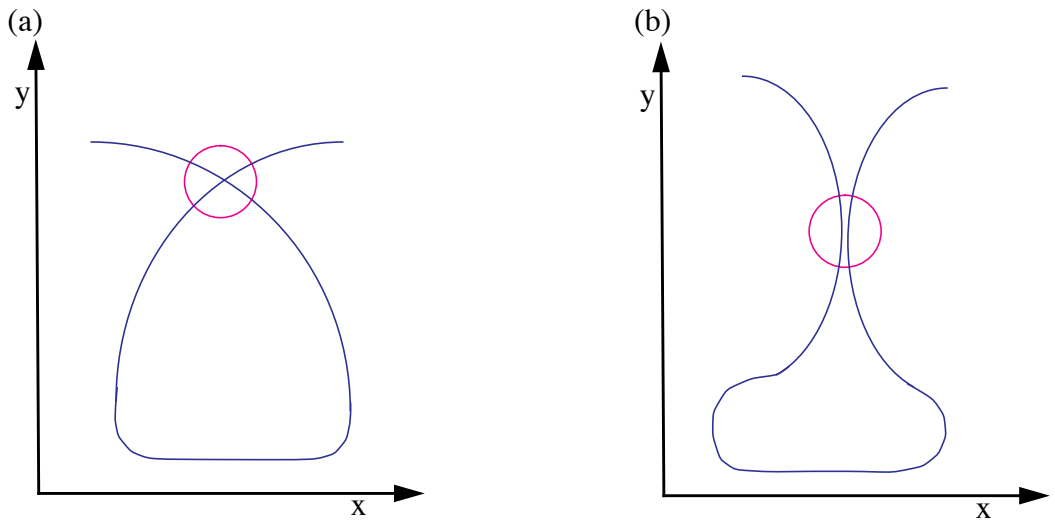

FIGURE 2. Possible cases of bifurcation points where the circle represents the actual step size. (a) The curve intersects itself; (b) The curve comes very “close” to itself.

$$\begin{aligned} &f(x_i, y_i) + f_x(x_i, y_i)dx\Delta s + f_y(x_i, y_i)dy\Delta s + \\ &f_{xx}(x_i, y_i)dx^2\Delta s^2 + 2f_{xy}(x_i, y_i)dxdy\Delta s^2 + f_{yy}(x_i, y_i)dy^2\Delta s^2 = 0 \end{aligned} \tag{9}$$

As already mentioned, the first three terms in the sum vanish, which leads to

$$f_{xx}(x_i, y_i)dx^2\Delta s^2 + 2f_{xy}(x_i, y_i)dxdy\Delta s^2 + f_{yy}(x_i, y_i)dy^2\Delta s^2 = 0 \tag{10}$$

In order to increase the readability of the following formulas, we use the following notation:

$$\begin{aligned} A &= f_{xx}(x_i, y_i) \\ B &= f_{xy}(x_i, y_i) \\ C &= f_{yy}(x_i, y_i) \end{aligned} \tag{11}$$

If we consider Equation (10) as a quadratic equation in the quotient q defined by dx/dy, we get the following solution:

$$q_{1,2} = \frac{dx}{dy} = \frac{-B \pm \sqrt{B^2 - AC}}{A} = \frac{C}{-B \mp \sqrt{B^2 - AC}} \tag{12}$$

To actually compute dx and dy, we substitute this result in Equation (3), which results in

$$\begin{pmatrix} dx \\ dy \end{pmatrix} = \pm \begin{pmatrix} \sqrt{\dfrac{1}{\dfrac{1}{q_{1,2}^2}+1}} \\ \sqrt{\dfrac{1}{q_{1,2}^2+1}} \end{pmatrix} = \pm \begin{pmatrix} \sqrt{\dfrac{C^2 - AC + 2B\,(B \pm \sqrt{B^2 - AC})}{(A-C)^2 + 4B^2}} \\ \sqrt{\dfrac{A^2 - AC + 2B\,(B \mp \sqrt{B^2 - AC})}{(A-C)^2 + 4B^2}} \end{pmatrix} \tag{13}$$

where any arbitrary combination of plus and minus from Equation (12) and Equation (13) results in four different distinct direction vectors (dx, dy). We will store all four directions in a list and follow their branches one at a time. This is explained in greater detail in Section 3.

We discussed the usual case of two intersection points and studied in detail what happens when these formulas fail when we encounter bifurcation points. Unfortunately, it could happen that we do not have four intersection points, but six, eight, or more. That means, that actually more then the two branches, as shown in Figure 1, meet at the same area. In that case, we realize that the second partial derivatives in Equation (9) vanishes also, leaving us with no choice other than including even higher order terms in the approximation of the Taylor series. This implies that we will have to solve higher order polynomials to produce the expected number of solutions.

2.3 The Corrector Step

Although we have tried hard to predict a new point p_p on the curve, it is not guaranteed to fall accurately on the curve. We can only hope to find a point that is "very close" to the curve. The goal of the *corrector step* is to find a "closest" point on the curve, starting at p_p. Among the many methods for finding zero points of functions, the Newton method is very stable and fast, and therefore very common. The Newton method is a numerical iteration method which converges to a zero point p^* of a function f, starting at a point p_0 that should be fairly close to the zero point already. In our case, the starting point for the Newton method will be p_p – the result of the predictor step.

The predictor step produced an initial value $p_0 = p_p = (x_p, y_p) = (x_0, y_0)$ for the iteration method. At each iteration step we take the new and improved value from the previous iteration and try to improve it even further, so that we actually end up on the curve itself. The result of the current iteration can be written as:

$$p_{k+1} = \begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k + \Delta x \\ y_k + \Delta y \end{pmatrix}, \tag{14}$$

where k marks the iteration number.

But how can we determine $(\Delta x, \Delta y)$? Here we exploit the same idea as we did in Euler's Method. Of course, we want that our new point (x_{k+1}, y_{k+1}) lies on the curve. So we add the condition that p_{k+1} is a curve point:

$$f(x_{k+1}, y_{k+1}) = 0. \tag{15}$$

The same ideas and manipulations as used for Equation (4) will lead us towards a solution. Substituting Equation (14) into Equation (15) and using a Taylor series expansion of f will result in:

$$f_x(x_k, y_k)\Delta x + f_y(x_k, y_k)\Delta y = -f(x_k, y_k). \tag{16}$$

That is, in fact, the formula for Newton's method for a two dimensional function. Since this is one equation with two unknowns, we have the freedom and duty to specify another condition, we want our iterate to satisfy. First of all, we want to make sure that our iteration process converges. Other than that, it would be nice if we could guarantee some sort of constant discretization of our curve. That is, we do not want the distance between samples to vary too much. Both of these ideas can be realized if we restrict our convergence to move in a right angle away from our previous iterate. To be more specific, the triangle formed by the points p_i, p_k and p_{k+1} should form a right angled triangle. Here, p_i is the initial point on the curve – the one we used in the predictor step to compute the point p_p. p_k and p_{k+1} are the previous and current iterates. In Figure 1 it was shown that this condition on our new iterate results in a somewhat circular movement around the initial point p_i. Because of this circular movement, we stay in a certain proximity to p_i, which increases the chances of convergence.

More formally, we want to add the condition that the vector $\overrightarrow{p_k p_i}$ is perpendicular to $\overrightarrow{p_k p_{k+1}}$. That is expressed in the following manner:

$$(p_i - p_k) \bullet (p_{k+1} - p_k) = \begin{bmatrix} x_i - x_k \\ y_i - y_k \end{bmatrix}^T \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \Delta x\,(x_i - x_k) + \Delta y\,(y_i - y_k) = 0 \tag{17}$$

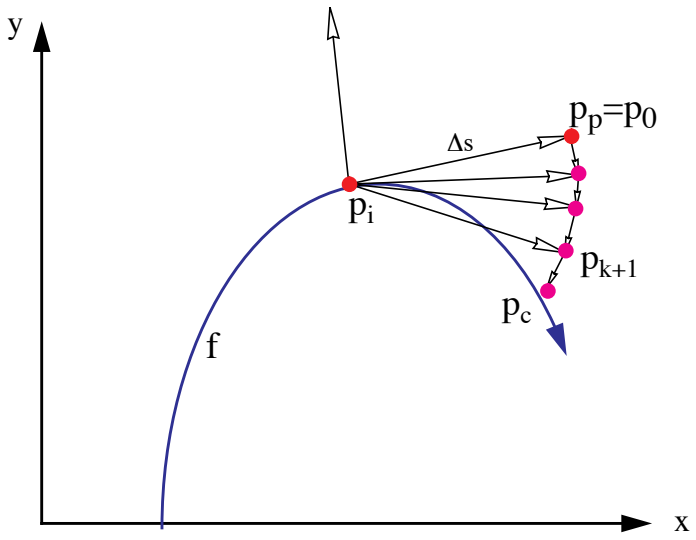

FIGURE 3. The corrector step. The Newton iterations and a forced circular movement of the iterates p_{k+1} converge towards an actual point on the curve p_c.

Equation (16) together with Equation (17) uniquely describe a new, corrected iterate p_{k+1}, which concludes the Corrector step. A geometric interpretation of these steps is illustrated in Figure 3.

3. The Algorithm

The pseudo code for our numerical algorithm is given in Figure 4. The algorithm starts by looking for a seed point (using a space subdivision method, as described in Section 3.1). The initial point will be most likely on some arbitrary part of the curve. Therefore, we will usually have two possible directions we can follow. We record both in a list for later retrieval (Line 3). At a bifurcation point a pixel will have four branches leaving it. In this case we will also record all possible branch directions in this list (Line 18). After retrieving a starting point for a new branch (Line 5) we trace the branch until we find another bifurcation point or until we reach the limits of our display device. In the case of a closed curve we also need to check when we get back to an already traced branch of the curve. A simple way to do this is explained in Section 3.5.

```
1. find an initial point p_i on the surface;
2. determine initial step size Δs;
3. record (p, Δs, bifurlist);                    /* for both initial direction */
4. repeat
5.     get p_i, Δs from bifurlist
6.     repeat                                    /* continue current branch */
7.         repeat                                /* find a reliable new point on the curve */
8.             p_p := Predictor(p_i)
9.             p_c := Corrector(p_p)
10.            C := approx_curvature(p_c, p_p)
11.            Δs := Δs/2
12.        until (p_c converges AND 1/(2C) < Δs) OR bifur_test(p_c)
13.        Δs := Δs*2;
14.        if 1/(2C) >Δs then
15.            increase Δs;
16.        draw_line(p_c, p_i);
17.        if bifur_test(p_c) then
18.            record(p_c, Δs, bifurlist)
19.        p_i := p_c;
20.    until (p_c is in the covered region or outside the screen OR bifur_test(p_c));
21. until bifurlist is empty
```

FIGURE 4. pseudo code for the curve tracking algorithm.

For each new predictor point we need to make sure that the corrector didn't fail (Line 7 through Line 12). Once we have gained confidence about our corrected point, we check whether we can adjust the step size (Line 15). We connect the previous point to the current point by a simple line (Line 16). If we found a new bifurcation point, we need to record it (Line 18) and quit this loop. In any other case we repeat this procedure and find another point on the same branch.

We now describe in more detail the implementation of some of the more involved segments of the algorithm.

3.1 Finding a Starting Point

If a seed point is not provided by the user, we employ a space subdivision approach. We divide the image region into four quadrants and compute the function at each corner of the quadrants. If the sign of the function is the same at all corners, we will apply the same subdivision to all four quadrants in a breath-first scheme. If we do find a sign change, we continue to subdivide this specific quadrant in a depth-first scheme up to the pixel- or even subpixel level, depending on how accurate our initial point should be.

3.2 Implementing the Corrector Step

Since the predicted point is only an approximation of a curve point, the corrector step is employed to converge to an actual point on the curve. Because of the approximations in our formulas, the Newton corrector step (explained in Section 2.3) usually needs to be repeated a few times. That is done within the Corrector procedure in Line 9. But how many times should we repeat the Newton step? Of course, we have to fight round off errors caused by the discrete computer arithmetic as well. The quality of our discretized curve greatly depends on our stopping criteria, as well as the successful completion of our algorithm. Therefore, we need to know when $f(p_{k+1})$ gets close enough to zero. The problem is that *close* means very different things for different functions. If we had more knowledge about the function, we could employ an absolute error test, like $f(p_{k+1}) < \varepsilon$. For a general algorithm, we will have no idea how small ε needs to be to assure spatial proximity. This notion of *being close* to the curve might also change at different ranges for the curve.

The method we chose adapts the value of ε and utilizes the fact that we draw to a finite grid. Every once in a while (e.g. every tenth time we call the Corrector procedure) we check the quality of the corrected point p_c. For this quality check we define a rectangular region centered at p_c and check for a sign change of the function f at the corner points. The size of this rectangular region can be a parameter L specified by the user or simply the current grid size. If there is a sign change in the corner points, we know that our curve will be at most $L/\sqrt{2}$ away. Therefore, we could relax our stopping criteria and set ε to $1.1|f(p_{k+1})|$. On the other hand, if we find that our corrected value isn't close enough to our curve, we should strengthen our stopping condition and set ε to $10^{-2}|f(p_{k+1})|$ and redo the corrector

step. This finally results in a relative (adaptive) error test, which is certainly superior than an absolute (static) error test.

3.3 Determining The Step Size Δs

The other crucial and very sensitive parameter of this algorithm is the step size Δs . One big advantage of our algorithm is the freedom to choose the step size. When the curve shows a small curvature we will choose a fairly large Δs , allowing the algorithm to step quickly through this “almost linear” part of the curve. On the other hand, when the curve changes rather quickly, i.e. the curvature of the function is high, we have to decrease Δs appropriately. This adaptive mechanism gives us the ability to capture the nature of our curve in greater detail when necessary.

The mathematical definition of the curvature is

$$C(p) = \lim_{q \to p} \frac{\alpha(p) - \alpha(q)}{\widehat{pq}}, \tag{18}$$

where $\alpha(p)$ denotes the angel of the tangent with the x-axis and $\widehat{pq}$ denotes the arc length between the two points. For implicit functions, C can be computed as

$$C = \frac{-f_y^2 f_{xx} + 2 f_x f_y f_{xy} - f_x^2 f_{yy}}{\left(f_x^2 + f_y^2\right)^{3/2}} \tag{19}$$

(See [6]). We compute first derivatives anyway for the Newton corrector, but would like to avoid computing second derivatives. Therefore, we use an approximation to the curvature that is based on Equation (18), where we use the current and previous point on the curve. Since we have the first derivatives already computed we can precisely compute the tangent angles. The arc length between the two points is simply approximated with the current step size. To compute the initial step size we could either use Equation (19) or just use the grid size. Choosing the initial step size for a bifurcation point can be a problem. If we are stepping too far, we might ‘loose track’ of the current branch. If we are stepping too close, the corrector might converge to a different, neighboring branch. Therefore we left it up to the user to choose this initial step size.

Another indicator of the quality of our step size would be the number of corrector steps we had to take before we converged to a curve point. In our predictor step we are basically following the tangent of the curve. If the curve is fairly linear the predicted point p_p will be almost on the curve, resulting in just very few corrector steps. If the curve changes its direction rather fast, p_p will not be the best guess and we expect more correction steps. Therefore, if we use at most one Newton step to find a point on the curve, we simply double the step size.

3.4 Detecting Bifurcation Points

Bifurcation points are points where the curve intersects itself. At these points the first derivatives of the function evaluate to zero. Of course, it will be very unlikely to find the mathematically exact bifurcation point (in case one exists), because of the numerical nature of the algorithm. Therefore, we need to keep watching the first derivatives of a function for regions where they are *close* to zero. We have seen already how difficult such a test can be (see Section 3.2). Therefore, whenever we find a sign change in one of the partial derivatives (from the previous to the current point) we have good reason to suspect that there might be a bifurcation point close by. As a result, we decrease our step size and repeat the last predictor/corrector step until the step size is smaller than a given size, which can be defined by the user. If there is still reason to believe that there might be a bifurcation point, we create a rectangular region around the current point on the curve. Similar to the test we performed in Section 3.2, we test whether there is a sign change in the partial derivatives f_x and f_y. If we find a sign change in both partial derivatives we conclude that we detected a bifurcation point. Otherwise, we conclude that we probably found a turning point of the curve.

3.5 Stopping Criteria

We assume that we deal with a closed curve. Therefore, we need to find out if and when we returned to a part of a branch we have traced already. If we do not include this exit test (in Line 20), we will trace the curve over and over again, ending up in an infinite loop.

Because of the nature of a numeric algorithm it is very unlikely to return to the exact same point from where we started. Therefore, we cannot just test for equality. We also do not want to test the proximity

of the new point p_c to every one of the computed values along the curve. That might be possible for the first few values, when the cost is still low, but certainly not for arbitrary curves, where we have to compute many points to reasonably approximate the curves shape. The other problem is again the meaning of "very close". To solve this problem, we maintain a counter that counts the number of times we attempt to draw a pixel that has been drawn already. When the algorithm computes a point on the curve that is mapped to a pixel that was not drawn before, we set the counter to zero. Every time we compute a new point on the curve within a pixel that was previously drawn, we increase a counter by one. If the counter's value exceeds some threshold, the algorithm stops pursuing this path.

TABLE 1. Performance results comparing our algorithm. The empty cells denote cases where Chandler's algorithm could not draw the curve.

Figure	# Function evaluation per pixel		# pixels per predictor	# pixels per predictor
	Our Method	Chandler		
5a	1.89	2.68	6.21	2.55
5b	2.01	2.80	6.79	3.20
5c	3.27	2.92	3.51	2.49
6a	2.88	–	6.66	5.04
6b	4.18	–	4.05	3.67
6c	4.55	–	2.38	2.30

4. Results

We applied our algorithm to many different functions with and without bifurcation points and found that they were drawn correctly in any image resolution of 128^2 to 512^2. For images of low resolution (32^2 and sometimes 64^2) the user may have to explicitly specify a smaller step size. We have chosen three very common functions without a bifurcation point (Figure 5) and three interesting functions with many bifurcation points (Figure 6). Some interesting statistics for each group of functions is summarized in Table 1 and compared to Chandlers algorithm [7]. The test cases where drawn onto a 5122 image. Drawing times, for all examples, was under 0.2 seconds on a Silicon Graphics Crimson (150Mhz R4400). In the table we also provide the average number of pixels per one predictor step (showing the adaptive step size), and the average number of correct steps per predictor.

In general, the number of function evaluations per pixel for low resolutions is expectably high, since the curvature of a curve considered within a pixel is larger than if the same curve is subdivided into smaller parts (i.e. for smaller pixel sizes). Therefore, the step size (in terms of the number of pixels) is smaller and the algorithm must perform more calculations on a per pixel basis. For high resolutions we have much fewer function evaluations per pixel than Chandler's algorithm without loss of quality. This demonstrates the power of the adaptive step size.

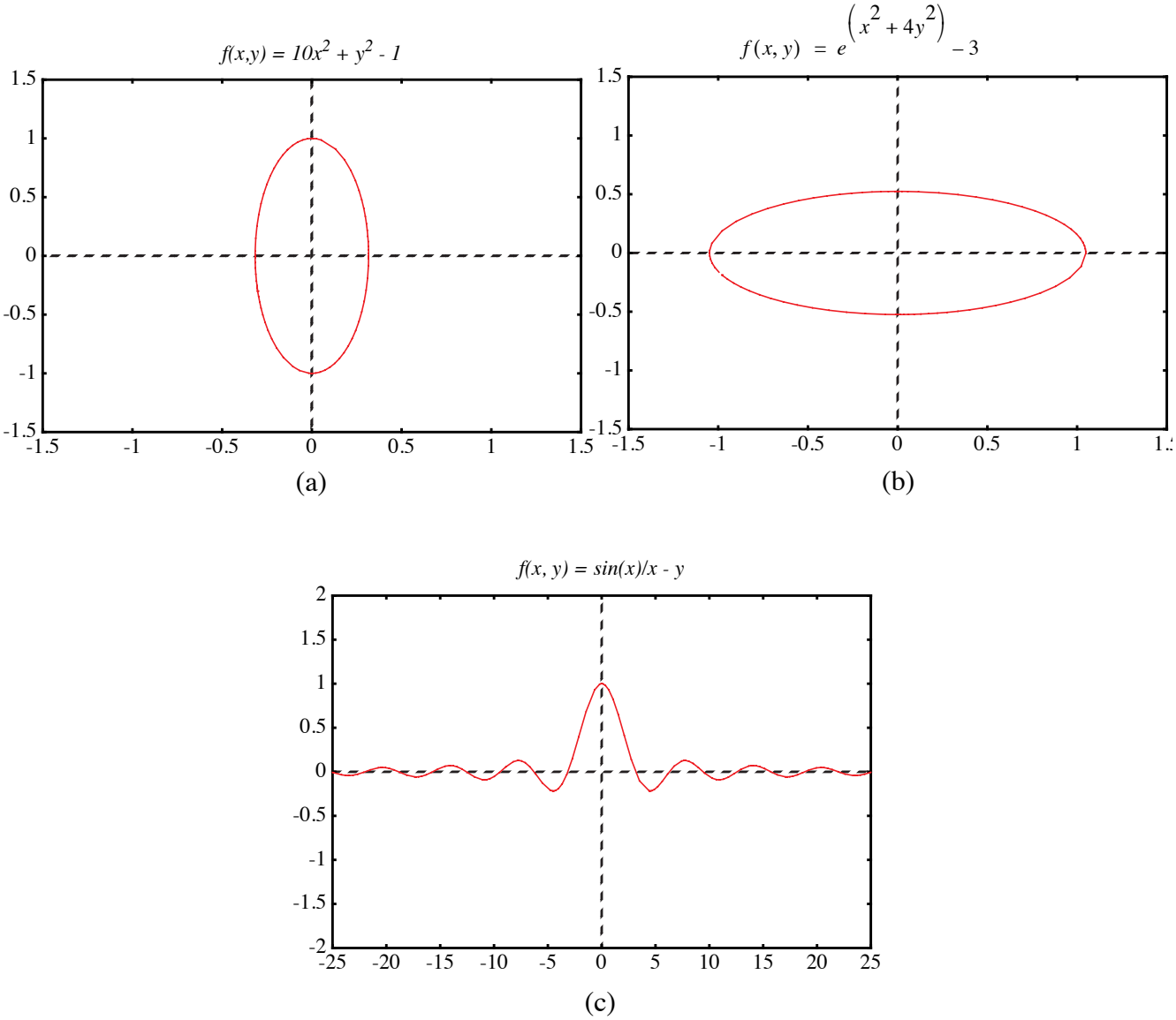

FIGURE 5. Curves without bifurcation points.

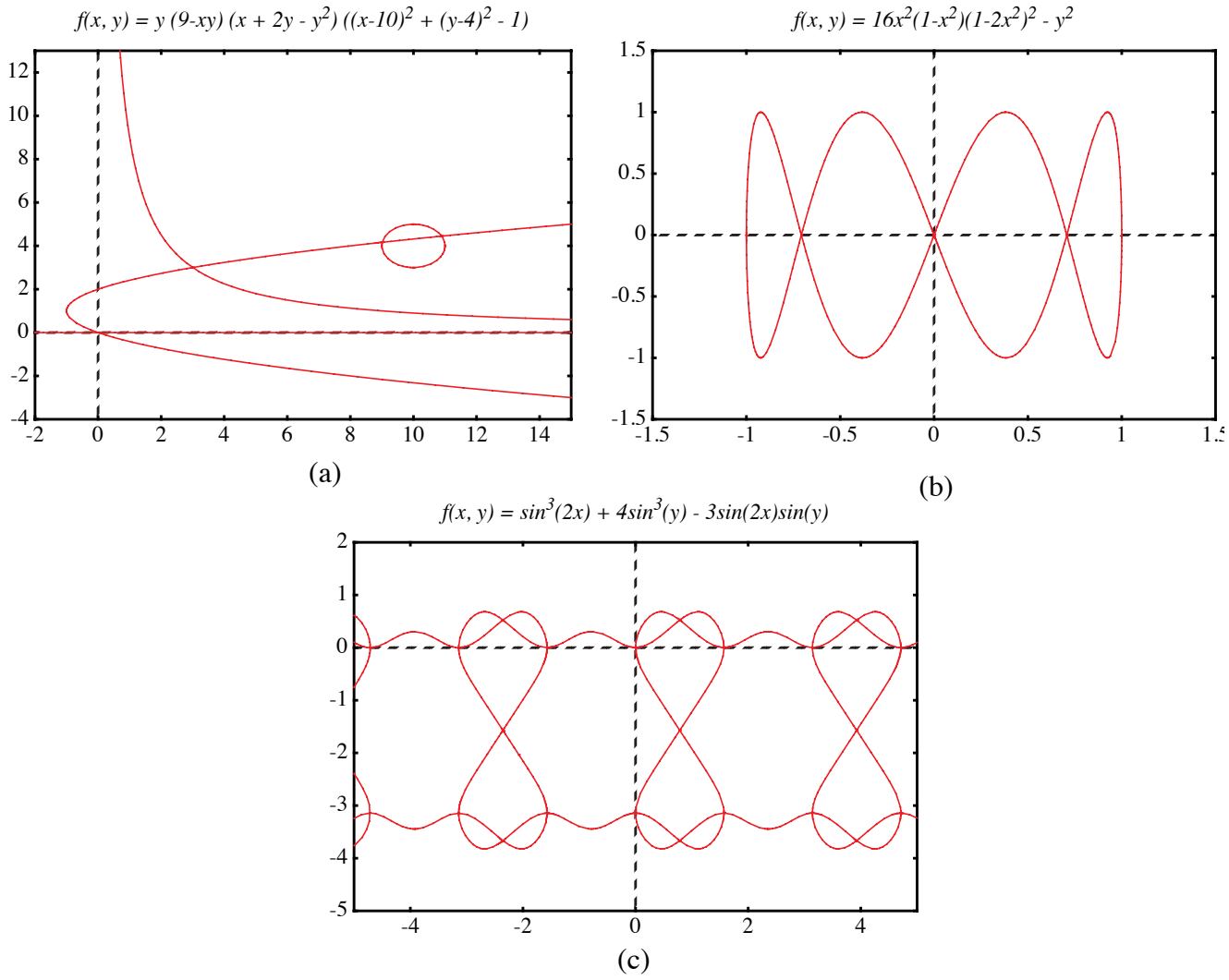

FIGURE 6. Curves with bifurcation points.

5. Conclusions

We presented here a numerical algorithm that is able to draw a large family of implicit functions. Through an adaptive step size design our performance is greatly optimized. We have shown that our approach outperforms previous tracking algorithms.

Our future research includes the improvement of the accuracy of the predictor step. Since we already know the derivatives at the previous point and, most of the time, also have this information available for the point before that, we could use the Hermite interpolation scheme and approximate our curve with a

polynom of third degree. This higher order approximation of our curve should result in fewer corrector steps. Since the higher order polynomial is computed only at the predictor step, saving several corrector steps, we expect a more efficient algorithm. A similar improvement can be made when connecting a new point on the curve with the previous one. Instead of a line through these point, one could use a higher order polynomial. Improvements to the accuracy of the corrector procedure are also possible, for example, by using a better corrector than Newton's. Finally, we plan to explore the extension of our methods to 3D voxelization of implicit surfaces.

6. References

[1] Allgower E. and S. Gnutzmann, "Simplicial Pivoting for Mesh Generation of Implicitly Defined Surfaces", Computer Aided Geometric Design, 8(4), 1991, pp. 30.

[2] Arvo J. and K. Novins, "Iso-Contour Volume Rendering", Proceedings of 1994 Symposium On Volume Visualization.

[3] Baker G. and E. Overman, "The Art of Scientific Computing", Draft Edition, The Ohio State Bookstore.

[4] Blinn J.F., "A Generalization of Algebraic Surface Drawing", ACM Transactions on Graphics, 1(3), 1982, pp. 235-256.

[5] Bloomenthal J., "Polygonization of Implicit Surfaces", Computer Aided Geometric Design, 5(4), 1988, pp. 341-356.

[6] Bronstein I. N. and Semendjajew, K. A. "Taschenbuch der Mathematik", BSB B.G.Teubner Verlagsgesellschaft, Leipzig, 1985.

[7] Chandler R.E., "A Tracking Algorithm for Implicitly Defined Curves", IEEE Computer Graphics and Applications, 8(2), March 1988, pp. 83-89.

[8] Foley J. D., A. van Dam, S. K. Feiner and J. F. Hughes, "Computer Graphics Principles and Practice", Addison-Wesley, 1990.

[9] Jordan B.W., W.J. Lennon, and B.D. Holm, "An Improved Algorithm for the Generation of Nonparametric Curves", IEEE Trans. Computers, Vol. C-22, 1973, pp. 1052-1060.

[10] Hall M. and J.Warren, "Adaptive Polygonization of Implicitly Defined Surfaces", IEEE Computer Graphics and Applications, 10(6), November 1990, pp. 33-42.

[11] Hanrahan P., "Ray Tracing Algebraic Surfaces", Computer Graphics (SIGGRAPH'83 Proceedings), 17(3), 1983, pp. 83-90.

[12] J.C. Hart. Ray Tracing Implicit Surfaces. WSU Technical Report EECS-93-014. Chapter in "Design, Visualization, and Animation of Implicit Surfaces," ACM SIGGRAPH '93 intermediate course notes, J. Bloomenthal and B. Wyvill, eds., Aug. 1993.

[13] Hobby J.D., "Rasterization of Nonparametric Curves", ACM Transactions on Graphics, 9(3), July 1990, pp. 262-277.

[14] Kalra D. and A.H. Barr, "Guaranteed Ray Intersections with Implicit Surfaces", Computer Graphics (SIGGRAPH'89 Proceedings), 23, 1989, pp. 297-306.

[15] Lorensen W. and H. Cline, “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”, Computer Graphics, 21(4), 1987, pp. 163-169.

[16] Sederberg T.W., D.C.Anderson, and R.N.Goldman, “Implicitization, Inversion, and Intersection of Planar Rational Cubic Curves”, Computer Vision, Graphics, and Image Processing, 31(1), 1985, pp. 89-102.

[17] Sederberg T.W. and A.K.Zundel, “Scan Line Display of Algebraic Surfaces”, Computer Graphics (SIGGRAPH’89 Proceedings), 23(4), 1989, pp. 147-156.