Robust Adaptive Approximation of Implicit Curves

HÉLIO LOPES[1] JOÃO BATISTA OLIVEIRA[2] LUIZ HENRIQUE DE FIGUEIREDO[3]

[1]Departamento de Matemática, Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente 225, 22453-900 Rio de Janeiro, RJ, Brazil
lopes@mat.puc-rio.br

[2]Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul
Avenida Ipiranga 6681, 90619-900 Porto Alegre, RS, Brazil
oliveira@inf.pucrs.br

[3]IMPA–Instituto de Matemática Pura e Aplicada
Estrada Dona Castorina 110, 22461-320 Rio de Janeiro, RJ, Brazil
lhf@impa.br

Abstract. We present an algorithm for computing a robust adaptive polygonal approximation of an implicit curve in the plane. The approximation is adapted to the geometry of the curve because the length of the edges varies with the curvature of the curve. Robustness is achieved by combining interval arithmetic and automatic differentiation.

Keywords: piecewise linear approximation; interval arithmetic; automatic differentiation; geometric modeling.

1 Introduction

An *implicit object* is defined as the set of solutions of an equation $f(p) = 0$, where $f\colon \Omega \subseteq \mathbf{R}^n \to \mathbf{R}$. For well-behaved functions f, this set is a surface of dimension $n-1$ in $\mathbf{R}^n$. Of special interest to computer graphics are *implicit curves* ($n = 2$) and *implicit surfaces* ($n = 3$), although several problems in computer graphics can be formulated as high-dimensional implicit problems [12, 17].

Applications usually need a geometric model of the implicit object, typically a polygonal approximation. While it is easy to compute polygonal approximations for parametric objects, computing polygonal approximations for implicit objects is a challenging problem for two main reasons: first, it is hard to find points on the implicit object [9]; second, it is hard to connect isolated points into a mesh [8].

In this paper, we consider the problem of computing a polygonal approximation for a curve $\mathcal{C}$ given implicitly by a function $f\colon \Omega \subseteq \mathbf{R}^2 \to \mathbf{R}$, that is,

$$\mathcal{C} = \{(x, y) \in \mathbf{R}^2 : f(x, y) = 0\}.$$

In Section 2 we review some methods for approximating implicit curves, and in Section 3 we show how to compute *robust adaptive* polygonal approximations. By "adaptive" we mean two things: first, Ω is explored adaptively, in the sense that effort is concentrated on the regions of Ω that are near $\mathcal{C}$; second, the polygonal approximation is adapted to the geometry of $\mathcal{C}$, having longer edges where $\mathcal{C}$ is flat and the curvature is low, and shorter edges where $\mathcal{C}$ bends more and the curvature is high. By "robust" we mean that the algorithm finds all pieces of $\mathcal{C}$ in Ω (implicit curves can have several connected components) and that it estimates correctly the variation of curvature along $\mathcal{C}$. In other words, the computed approximation captures efficiently both the topology and the geometry of $\mathcal{C}$. Adaption and robustness are achieved by combining interval arithmetic and automatic differentiation, which are explained in Section 3.

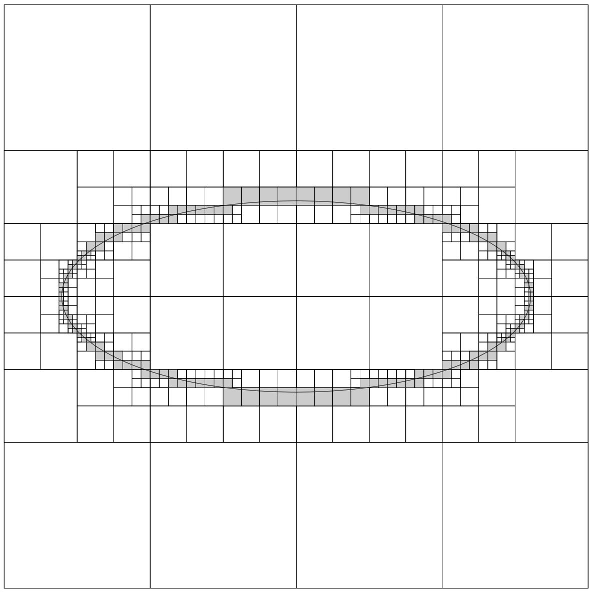


Figure 1: Our algorithm in action.

An example of what our algorithm does is shown in Figure 1 for the ellipse given implicitly by $x^2/6 + y^2 = 1$. Further examples are given in Section 3.6 (see Figure 3).

2 Approximation methods for implicit curves

Approximating a curve $\mathcal{C}$ given implicitly by $f\colon \Omega \to \mathbf{R}$ is a hard problem mainly because it is hard to find just *where* in Ω that $\mathcal{C}$ lies: it can be anywhere, and even nowhere, in Ω, and it can have several components, of varying size, some of which may be closed and nested inside each other.

The classical method for avoiding chasing $\mathcal{C}$ blindly inside Ω is to decompose Ω into a grid of small rectangular or triangular cells and then traverse the grid and locate $\mathcal{C}$ by identifying those cells that intersect $\mathcal{C}$. This process is called *enumeration*.

The two main problems with enumeration are: how to select the resolution of the grid, so that we do not miss small components of $\mathcal{C}$; and how to decide whether a cell in the grid intersects $\mathcal{C}$. One simple test for the second problem is to check the sign of f at the vertices of the cell. If these signs are not all equal, then the cell must intersect $\mathcal{C}$ (provided f is continuous, of course). However, if the signs are the same, then we cannot discard the cell, because it might contain a small closed component of $\mathcal{C}$ in its interior, or $\mathcal{C}$ might enter and leave the cell through the same edge.

In practice, the simplest solution to both problems is to use a fine regular grid and hope for the best. Figure 2 shows an example of such *full enumeration* on a regular rectangular grid. The enumerated cells are shown in grey. The points where $\mathcal{C}$ intersects the boundary of those cells can be computed by linear interpolation or, if higher accuracy is desired, by any other classical method, such as bisection. Note that the output of an enumeration is simply a set of line segments; some post-processing is needed to arrange these segments into polygonal lines.

Full enumeration works well — provided a fine enough grid is used — but it can be very expensive, because many cells in the grid will not intersect $\mathcal{C}$, specially if $\mathcal{C}$ has components of different sizes (as in Figure 2). If we take the number of evaluations of f as a measure of the cost of the algorithm, then full enumeration will waste many evaluations on cells that are far away from $\mathcal{C}$. Typically, if the grid has $O(n^2)$ cells, then only $O(n)$ cells will intersect $\mathcal{C}$. The finer the grid, the more expensive full enumeration is.

Another popular approach to approximating an implicit curve is *continuation*, which starts at a point on the curve and tries to step along the curve. One simple continuation method is to integrate the Hamiltonian vector field $(-\partial f/\partial y, \partial f/\partial x)$, combining a simple numerical integration method with a Newton corrector [2]. Another method is to follow the curve across the cells of a regular cellular decomposition of Ω by pivoting from one cell to another, without having to compute the whole decomposition [12].

Continuation methods are attractive because they concentrate effort where it is needed, and may adapt the computed approximation to the local geometry of the curve, but they need starting points on each component of the curve; these points are not always available and may need to be hunted. Moreover, special care is needed to handle closed components correctly.

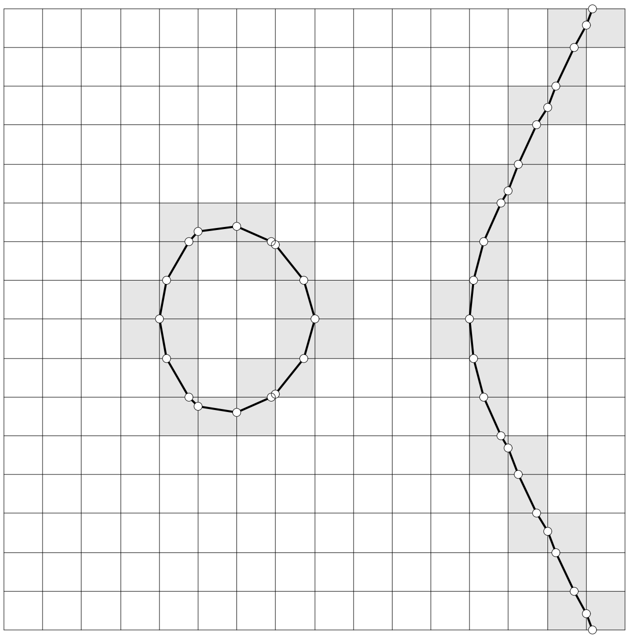


Figure 2: Full enumeration of the curve given implicitly by $y^2 - x^3 + x = 0$ in the square $\Omega = [-2, 2] \times [-2, 2]$.

What we need is an efficient and robust method that performs *adaptive enumeration*, in which the cells are larger away from the curve and smaller near it, so that computational effort is concentrated where it is most needed. The main obstacle in this approach is how to decide *reliably* whether a cell is away from the curve. Fortunately, interval methods provide a robust solution for this problem, as explained in Section 3. Moreover, by combining interval arithmetic with automatic differentiation (also explained in Section 3), it is possible to reliably estimate the curvature of $\mathcal{C}$ and thus adapt the enumeration not only *spatially*, that is, with respect to the location of $\mathcal{C}$ in Ω, but also *geometrically*, by identifying large cells where $\mathcal{C}$ can be approximated well by a straight line segment. The goal of this paper is to present a method for doing exactly this kind of completely adaptive approximation, in a robust way.

3 Robust adaptive polygonal approximation

As discussed in Section 2, what we need for robust adaptive enumeration is some kind of *oracle* that *reliably* answers the question "Does this cell intersect $\mathcal{C}$?". Testing the sign of f at the vertices of the cell is an oracle, but not a reliable one. It turns out that it is easier to implement oracles that reliably answer the complementary question "Is this cell away from $\mathcal{C}$?". Such oracles test the *absence* of $\mathcal{C}$ in the cell, rather than its presence, but they are just as effective for reliable enumeration. We shall now describe how such *absence oracles* may be implemented and how to use them to compute adaptive enumerations reliably.

3.1 Inclusion functions and adaptive enumeration

An absence oracle for a curve $\mathcal{C}$ given implicitly by $f\colon \Omega \subseteq \mathbf{R}^2 \to \mathbf{R}$ can be readily implemented if we have an *inclusion function* for f, that is, a function F defined on the subsets X of Ω and taking real intervals as values such that

$$F(X) \supseteq f(X) = \{f(x,y) : (x,y) \in X\}.$$

In words, $F(X)$ is an *estimate* for the *complete* set of values taken by f on X. This estimate is not required to be tight: $F(X)$ may be strictly larger than $f(X)$. Nevertheless, even if not tight, estimates provided by inclusion functions are sufficient to implement an absence oracle: if $0 \notin F(X)$, then $0 \notin f(X)$, that is, $f(x,y) \neq 0$ for *all* points (x,y) in X, and X does not intersect $\mathcal{C}$. Note that this is *not* an approximate statement: $0 \notin F(X)$ is a *proof* that X does not intersect $\mathcal{C}$.

Once we have a reliable absence oracle, it is simple to write a reliable adaptive enumeration algorithm as follows:

```
explore(X):
    if 0 ∉ F(X) then
        discard X
    elseif diam(X) < ε then
        output X
    else
        divide X into smaller pieces X_i
        for each i, explore(X_i)
```

Starting with a call to explore(Ω), this algorithm performs a recursive exploration of Ω, discarding subregions X of Ω when we can *prove* that X does not contain any part of the curve $\mathcal{C}$. The recursion stops when X is smaller than a user-selected tolerance ε. The output of the algorithm is a list of small cells whose union is *guaranteed* to contain the curve $\mathcal{C}$.

In practice, Ω is a rectangle and X is divided into rectangles too. A typical choice is to divide X into four equal rectangles, thus generating a quadtree [27,28], but it is also common to bisect X perpendicularly to its longest size, or to alternate the directions of the cut.

3.2 An algorithm for adaptive approximation

The algorithm above is only *spatially* adaptive, because all output cells have the same size. Geometric adaption requires that we estimate how the curvature of $\mathcal{C}$ varies inside a cell. This can be done by using an inclusion function G for the (normalized) gradient of f, because this gradient is normal to $\mathcal{C}$. Note that G has two components: one for $\partial f/\partial x$ and one for $\partial f/\partial y$; the value $G(X)$ is thus a rectangle (i.e., the product of two intervals). When $G(X)$ is small, the normal to $\mathcal{C}$ is not changing much inside X, and so $\mathcal{C}$ is approximately flat in X, which means that $\mathcal{C}$ can be approximated well by a straight line segment in X.

With the inclusion of estimates for the curvature and an additional user-selected tolerance δ, the algorithm above changes from adaptive enumeration to adaptive approximation with line segments:

```
explore(X):
    if 0 ∉ F(X) then
        discard X
    elseif diam(X) < ε or diam(G(X)) < δ then
        approx(X)
    else
        divide X into smaller pieces X_i
        for each i, explore(X_i)
```

This algorithm works essentially like to previous one, except that now approx(X) is called to approximate $\mathcal{C}$ inside X by a segment. This is done by computing and joining the points where $\mathcal{C}$ intersects the boundary of X.

Finding these intersection points reduces to finding the zero of a single-variable real function h in an interval $[a,b]$ such that the signs of $h(a)$ and $h(b)$ are different. Any classical method can be used for this, such as bisection, Newton's method, or Brent's method. The important point is to use the same method everywhere and to arrange the computation such that it gives the *same* result at neighboring cells, even if those cells do not share complete edges. If $\mathcal{C}$ only cuts cell edges once, then these precautions guarantee that the computed line segments can be glued together into a consistent polygonal approximation for $\mathcal{C}$. Linear interpolation is *not* recommended for computing those zeros, because it will probably generate inconsistent results.

We shall now briefly describe *interval arithmetic*, the natural tool for implementing inclusion functions, and *automatic differentiation*, the natural tool for computing gradients.

3.3 Interval arithmetic

Interval arithmetic was originally introduced as a tool to improve the reliability of numerical computations through automatic error control [24]. Since then, it has been recognized as the simplest technique for studying the global behaviour of real functions, and as the natural tool for implementing inclusion functions. Interval arithmetic has been used successfully in several graphics problems [5,11,13,22, 25,29,35], including the implementation of absence oracles for adaptive enumeration of implicit curves [23,30,33].

Interval arithmetic can compute robust bounds for the range of functions by representing ranges of values as intervals and providing interval versions of all basic arithmetic operations and elementary functions. Interval operations must generate intervals that are guaranteed to contain *all* the values obtained by operating with *all* the numbers in the input intervals. This is easy to do for the elementary operations and functions [24].

Implementing interval arithmetic in floating-point machine arithmetic is not difficult, although care has to be taken with roundings [31]. There are several packages for interval arithmetic available in the Internet [20]. Specially convenient are packages in languages that allow operator overloading, such as C++ and Fortran 90, because then algebraic expressions can be written in the familiar way and inclusion functions are automatically built by the compiler. When operator overloading is not available, inclusion functions can be built with the help of a precompiler [7].

3.4 Automatic differentiation

To estimate how the curvature of the implicit curve $\mathcal{C}$ varies inside a cell X of Ω, we need to compute and estimate the gradient of f inside X. We have seen that interval arithmetic is a good tool for estimating functions, but how do we *compute* the gradient?

One answer is to use *symbolic differentiation*, which manipulates an algebraic expression for f into algebraic expressions for $\partial f/\partial x$ and $\partial f/\partial y$. Another answer is to use *numerical differentiation*, which computes *approximations* for $\partial f/\partial x$ and $\partial f/\partial y$ based on Newton quotients or higher-order divided differences. Both techniques are fairly simple to implement, but they are not good solutions in general: Symbolic differentiation can generate very long expressions, specially when the expression for f contains many common sub-expressions — the evaluation of the derivative expressions thus obtained becomes slow. Numerical derivatives are fast to compute but notoriously ill-conditioned, and are best avoided.

It happens that there is a computational technique that combines the speed of numerical differentiation with the accuracy of symbolic differentiation: it is called *automatic* or *computational* differentiation. This simple technique has been rediscovered many times [19,24,26,36], but is still not well known; in particular, applications of automatic differentiation in computer graphics are still not common [21].

Derivatives computed with automatic differentiation are *not* approximate: the only errors in their evaluation are round-off errors, and these will be significant only when they are significant for evaluating the function itself.

Like interval arithmetic, automatic differentiation is easy to implement [18, 36]: instead of operating with single numbers, we operate with tuples of numbers $(u_0, u_1, \ldots, u_n)$, where u_0 is the value of the function and u_i is the value of its partial derivative with respect to the i-th variable. We extend the elementary operations and functions to these tuples by means of the chain rule and the elementary calculus formulas. Once this is done, derivatives are automatically computed for complicated expressions simply by following the rules for each elementary operation or function that appears in the evaluation of the function itself. In other words, any sequence of elementary operations for evaluating $f(x_1, \ldots, x_n)$ can be automatically transformed into a sequence of tuple operations that computes not only the value of f at a point $(x_1, \ldots, x_n)$ but also all the partial derivatives of f at this point. Again, operator overloading simplifies the implementation and use of automatic differentiation, but it can be easily implemented in any language [18], perhaps aided by a precompiler [7].

Here are some sample automatic differentiation formulas for $n = 2$:

$$\begin{aligned}
(u_0, u_1, u_2) + (v_0, v_1, v_2) &= (u_0 + v_0, u_1 + v_1, u_2 + v_2) \\
(u_0, u_1, u_2) \cdot (v_0, v_1, v_2) &= (u_0 v_0, u_0 v_1 + v_0 u_1, u_0 v_2 + v_0 u_2) \\
\sin(u_0, u_1, u_2) &= (\sin u_0, u_1 \cos u_0, u_2 \cos u_0) \\
\exp(u_0, u_1, u_2) &= (\exp u_0, u_1 \exp u_0, u_2 \exp u_0)
\end{aligned}$$

Note how values on the left-hand side are reused in the computation of partial derivatives on the right-hand side. This makes automatic differentiation much more efficient than symbolic differentiation: several common sub-expressions are identified and evaluated only once.

We can take the formulas for automatic differentiation and interpret them over intervals: each u_i is now an interval, and the operations on them are interval operations. This combination of automatic differentiation with interval arithmetic allows us to compute *interval estimates of partial derivatives* automatically, and is the last tool that we needed for the adaptive approximation algorithm of Section 3.2.

3.5 Implementation details

We implemented the algorithm described in Section 3.2 in C++, coding interval arithmetic routines from scratch and taking the automatic differentiation routines from the book by Hammer et al. [15].

To test whether the curve is flat in a cell X, we computed an interval estimate for the normalized gradient of f inside X. This gave a rectangle $G(X)$ in $[-1, 1] \times [-1, 1]$. The test $\operatorname{diam}(G(X)) < \delta$ was implemented by testing whether both sides of $G(X)$ were smaller than δ. This is not the only possibility, but it is simple and worked well, except for the non-obvious choice of the gradient tolerance δ.

Our implementation of $\mathsf{approx}(X)$ computed the intersection of $\mathcal{C}$ with a rectangular cell X by dividing X along its main diagonal into two triangles, and using classical bisection on the edges for which the sign of f at the vertices was different. As mentioned in Section 3.2, this produces a consistent polygonal approximation, even at adjacent cells that do not share complete edges.

If the sign of f was the same at all the vertices of X, then we simply ignored X; this worked well for the examples we used. If necessary, the implementation of approx may be refined by using the estimate $G(X)$ to test whether the gradient of f or one of its components is zero inside X. If these tests fail, then X can be safely discarded because X cannot contain small closed components of $\mathcal{C}$ and $\mathcal{C}$ cannot intersect an edge of X more than once: closed components must contain a singular point of f, and double intersections imply that $\partial f/\partial x$ or $\partial f/\partial y$ vanish in X. We did not find these additional tests necessary in our experiments.

3.6 Examples of adaptive approximation

Figure 3 shows several examples of adaptive approximations computed with our program. The white cells of many different sizes reflect the spatial adaption. The grey cells of many different sizes reflect the geometric adaption. Inside each grey cell, the curve is approximated by a segment.

The last two curves have complicated expressions, and the quadtree is too refined near the curves, making it hard to see the polygonal approximation. For this reason, we only show the leaves of the quadtree in these two examples.

4 Related work

Early work on implicit curves in computer graphics concentrated on rendering, and consisted mainly of continuation methods in image space. Aken and Novak [1] showed how Bresenham's algorithm for circles can be adapted to render more general curves, but only gave details for conics. Their work was later expanded by Chandler [6]. These two papers contain several references to the early work on the rendering problem. More recently, Glassner [14] discussed in detail a continuation algorithm for rendering.

Special robust algorithms have been devised for *algebraic curves*, that is, implicit curves defined by a *polynomial* equation. One early rendering algorithm was proposed by Aron [3], who computed the topology of the curve using the cylindrical algebraic decomposition technique from computer algebra. He also described a continuation algorithm that integrates the Hamiltonian vector field, but is guided by the topological structure previously computed. More recently, Taubin [34] gave a robust algorithm for rendering a plane algebraic curve. He showed how to compute constant-width renderings by approximating the Euclidean distance to the curve. His work can be seen as a specialized interval technique for polynomials.

Dobkin et al. [12] described in detail a continuation method for approximating implicit curves with polygonal lines. Their algorithm follows the curve across a regular triangular grid that is never fully built, but is instead traversed from one intersecting cell to another by reflection rules. Since the grid is regular, their approximation is not geometrically adaptive. Moreover, the selection of the grid resolution is left to the user and so the aliasing problems mentioned in Section 2 may still occur.

Suffern [32] seems to have been the first to try to replace full enumeration with adaptive enumeration. He proposed a quadtree exploration of the ambient space guided by two parameters: how far to divide the domain without trying to identify intersecting cells, and how far to go before attempting to approximate the curve in the cell. This heuristic method seems to work well, but of course its success depends on the selection of those two parameters, which must be done by trial and error.

Shortly afterwards, Suffern and Fackerell [33] applied interval methods for the robust enumeration of implicit curves, and gave an algorithm that is essentially the one described in Section 3.1. Their work is probably the first application of interval arithmetic in graphics (the early work of Mudur and Koparkar [25] seems to have been largely ignored until then).

In a course at SIGGRAPH'91, Mitchell [23] revisited the work of Suffern and Fackerell [33] on robust adaptive enumeration of implicit curves, and helped to spread the word on interval methods for computer graphics. He also described automatic differentiation and used it in ray tracing implicit surfaces.

Snyder [29, 30] described a complete modeling system based on interval methods, and included an approximation algorithm for implicit curves that incorporated a global parametrizability criterion in the quadtree decomposition. This allowed his algorithm to produce an enumeration that has final cells of varying size, but the resulting approximation is not adapted to the curvature.

Hickey et al. [16] described a robust program based on interval arithmetic for plotting implicit curves and relations.

Figueiredo and Stolfi [10] showed that adaptive enumerations can be computed more efficiently by using tighter interval estimates provided by affine arithmetic.

5 Conclusion

We have described an algorithm for robust adaptive approximation of implicit curves. As far as we know, this is the first algorithm which computes a reliable enumeration that is *both* spatially and geometrically adaptive.

The natural next step in this research is to attack implicit surfaces, which have recently become again an active research area [4]. The ideas and techniques presented in this paper are useful for computing robust adaptive approximations of implicit surfaces. However, the solution will probably be more complex, because we face more difficult topological problems, not only for the surface itself but also in the local approximation by plane polygons.

Acknowledgements. This research was done while J. B. Oliveira was visiting the Visgraf laboratory at IMPA during IMPA's summer post-doctoral program. Visgraf is sponsored by CNPq, FAPERJ, FINEP, and IBM Brasil. H. Lopes is a member of the Matmidia laboratory at PUC-Rio. Matmidia is sponsored by FINEP, PETROBRAS, CNPq, and FAPERJ. L. H. de Figueiredo is a member of Visgraf and is partially supported by a CNPq research grant.

References

[1] J. V. Aken and M. Novak. Curve-drawing algorithms for raster displays. *ACM Transactions on Graphics*, 4(2):147–169, 1985. Corrections in *ACM TOG*, 6(1):80, 1987.

[2] E. L. Allgower and K. Georg. *Numerical Continuation Methods: An Introduction*. Springer-Verlag, 1990.

[3] D. S. Arnon. Topologically reliable display of algebraic curves. *Computer Graphics*, 17(3):219–227, 1983 (SIGGRAPH'83).

[4] R. J. Balsys and K. G. Suffern. Visualisation of implicit surfaces. *Computers & Graphics*, 25(1):89–107, 2001.

[5] W. Barth, R. Lieger, and M. Schindler. Ray tracing general parametric surfaces using interval arithmetic. *The Visual Computer*, 10(7):363–371, 1994.

[6] R. E. Chandler. A tracking algorithm for implicitly defined curves. *IEEE Computer Graphics and Applications*, 8(2):83–89, 1988.

[7] F. D. Crary. A versatile precompiler for nonstandard arithmetics. *ACM Transactions on Mathematical Software*, 5(2):204–217, 1979.

[8] L. H. de Figueiredo and J. Gomes. Computational morphology of curves. *The Visual Computer*, 11(2):105–112, 1995.

[9] L. H. de Figueiredo and J. Gomes. Sampling implicit objects with physically-based particle systems. *Computers & Graphics*, 20(3):365–375, 1996.

[10] L. H. de Figueiredo and J. Stolfi. Adaptive enumeration of implicit surfaces with affine arithmetic. *Computer Graphics Forum*, 15(5):287–296, 1996.

[11] J. B. S. de Oliveira and L. H. de Figueiredo. Robust approximation of offsets and bisectors of plane curves. In *Proceedings of SIBGRAPI 2000*, pages 139–145. IEEE Press, October 2000.

[12] D. P. Dobkin, S. V. F. Levy, W. P. Thurston, and A. R. Wilks. Contour tracing by piecewise linear approximations. *ACM Transactions on Graphics*, 9(4):389–423, 1990.

[13] T. Duff. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. *Computer Graphics*, 26(2):131–138, 1992 (SIGGRAPH'92).

[14] A. Glassner. Andrew Glassner's Notebook: Going the distance. *IEEE Computer Graphics and Applications*, 17(1):78–84, 1997.

[15] R. Hammer, M. Hocks, U. Kulisch, and D. Ratz. *C++ Numerical toolbox for verified computing*. Springer-Verlag, Berlin, 1995.

[16] T. J. Hickey, Z. Qju, and M. H. V. Emden. Interval constraint plotting for interactive visual exploration of implicitly defined relations. *Reliable Computing*, 6(1):81–92, 2000.

[17] C. M. Hoffmann. A dimensionality paradigm for surface interrogations. *Computer Aided Geometric Design*, 7(6):517–532, 1990.

[18] M. Jerrell. Automatic differentiation using almost any language. *ACM SIGNUM Newsletter*, 24(1):2–9, Jan. 1989.

[19] H. Kagiwada, R. Kalaba, N. Rasakhoo, and K. Spingarn. *Numerical derivatives and nonlinear analysis*. Plenum Press, New York, 1986.

[20] V. Kreinovich. Interval software. `http://cs.utep.edu/interval-comp/intsoft.html`.

[21] D. Mitchell and P. Hanrahan. Illumination from curved reflectors. *Computer Graphics*, 26(2):283–291, 1992 (SIGGRAPH'92).

[22] D. P. Mitchell. Robust ray intersection with interval arithmetic. In *Proceedings of Graphics Interface'90*, pages 68–74, May 1990.

[23] D. P. Mitchell. Three applications of interval analysis in computer graphics. In *Frontiers in Rendering course notes*, pages 14-1–14-13. SIGGRAPH'91, July 1991.

[24] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[25] S. P. Mudur and P. A. Koparkar. Interval methods for processing geometric objects. *IEEE Computer Graphics & Applications*, 4(2):7–17, 1984.

[26] L. B. Rall. The arithmetic of differentiation. *Mathematics Magazine*, 59(5):275–282, 1986.

[27] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 1990.

[28] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[29] J. M. Snyder. *Generative Modeling for Computer Graphics and CAD*. Academic Press, 1992.

[30] J. M. Snyder. Interval analysis for computer graphics. *Computer Graphics*, 26(2):121–130, 1992 (SIGGRAPH'92).

[31] J. Stolfi and L. H. de Figueiredo. *Self-Validated Numerical Methods and Applications*. Monograph for 21st Brazilian Mathematics Colloquium, IMPA, Rio de Janeiro, 1997. Available at `ftp://ftp.tecgraf.puc-rio.br/pub/lhf/doc/cbm97.ps.gz`.

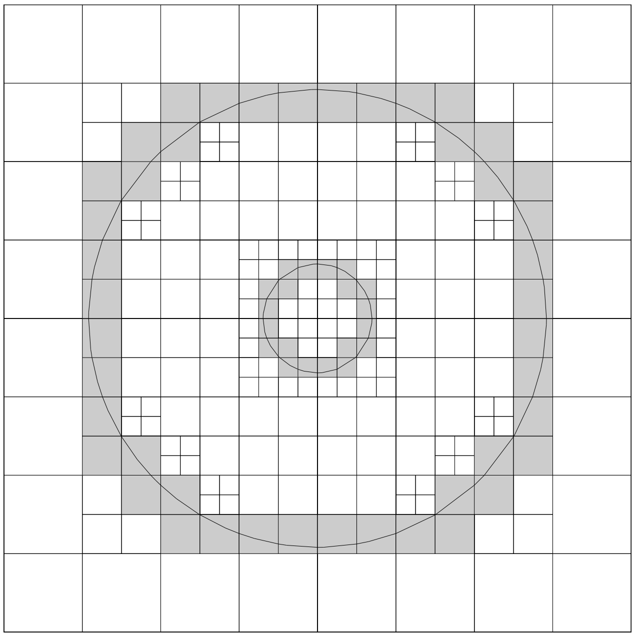
[32] K. G. Suffern. Quadtree algorithms for contouring functions of two variables. *The Computer Journal*, 33(5):402–407, 1990.

[33] K. G. Suffern and E. D. Fackerell. Interval methods in computer graphics. *Computers & Graphics*, 15(3):331–340, 1991.

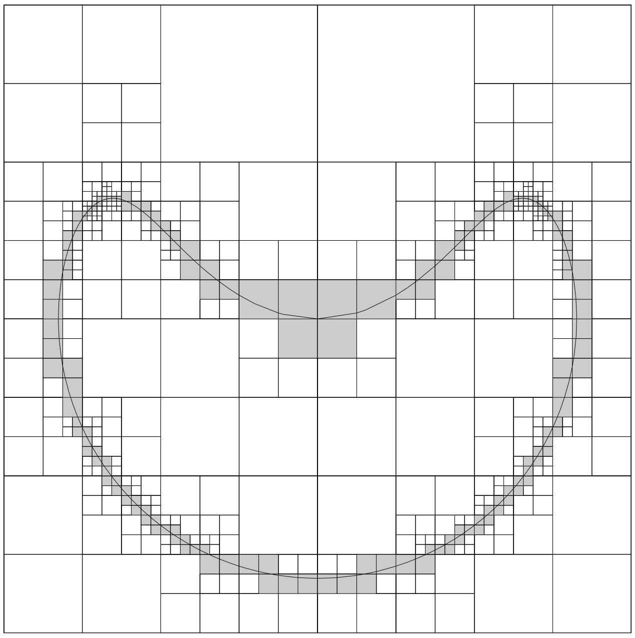
[34] G. Taubin. Rasterizing algebraic curves and surfaces. *IEEE Computer Graphics and Applications*, 14(2):14–23, 1994.

[35] D. L. Toth. On ray tracing parametric surfaces. *Computer Graphics*, 19(3):171–179, 1985 (SIGGRAPH'85).

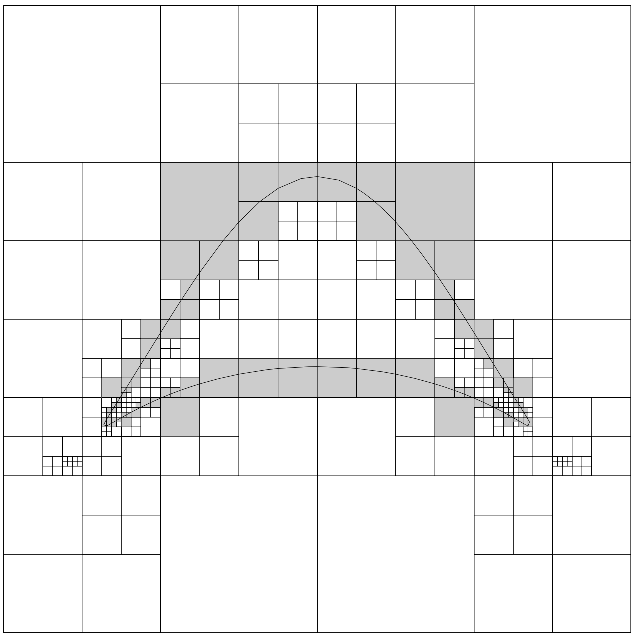
[36] R. E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.



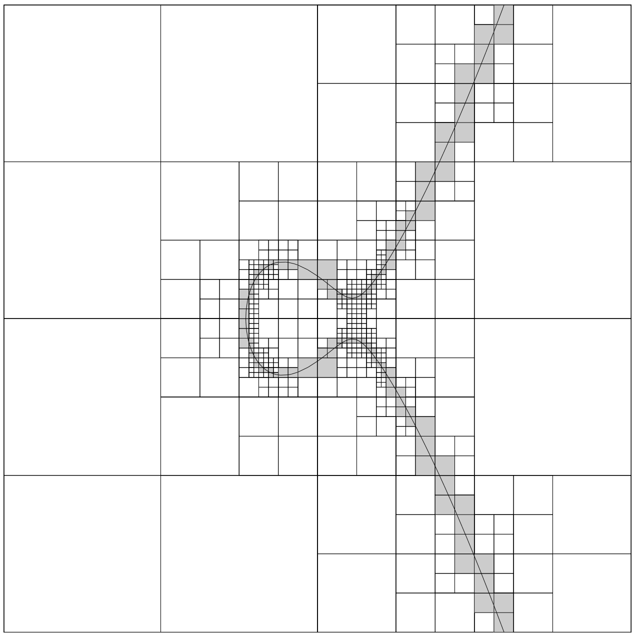
Two circles: $(x^2 + y^2)(1 - \sqrt{x^2 + y^2}) - 0.04 = 0$
$(L = 1.31, \quad \delta = 0.9)$



"Clown smile": $(y - x^2 + 1)^4 + (x^2 + y^2)^4 - 1 = 0$
$(L = 1.21, \quad \delta = 0.5)$



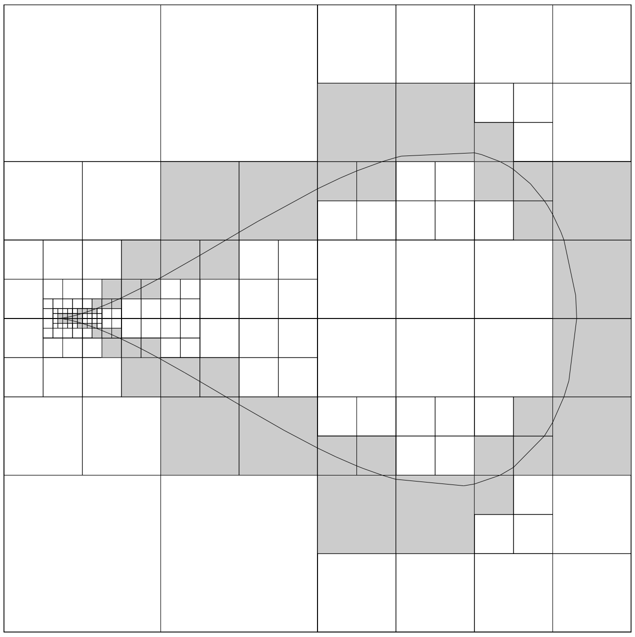
Bicorn: $y^2(a^2 - x^2) - (x^2 + 2ay - a)^2 = 0, \quad a = 0.75$
$(L = 1.1, \quad \delta = 0.8)$



Cubic: $y^2 - x^3 + x - 0.5 = 0$
$(L = 5.21, \quad \delta = 0.35)$

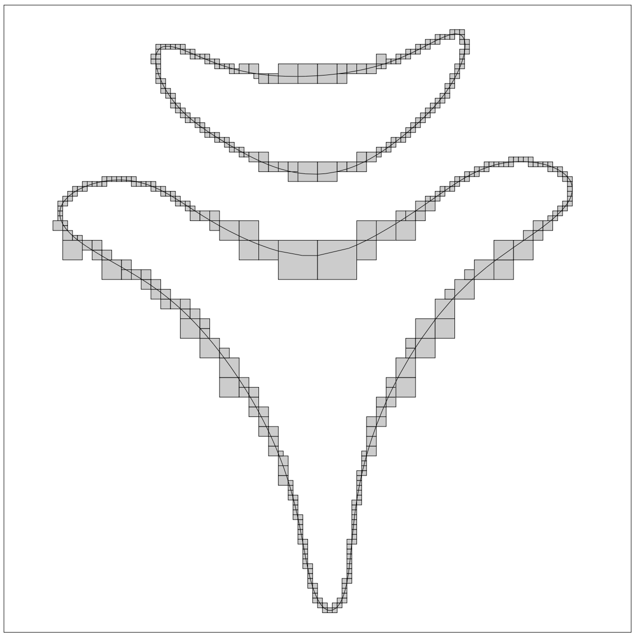
Figure 3: Examples of adaptive approximation.

The region is $\Omega = [-L, L] \times [-L, L]$ and the tolerance for gradient estimates is δ.



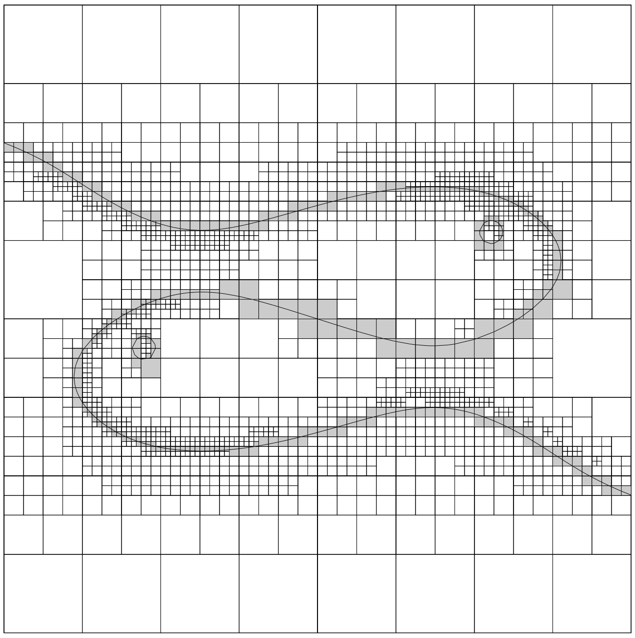
Pear: $4y^2 - (x+1)^3(1-x) = 0$

$(L = 1.21, \quad \delta = 0.85)$



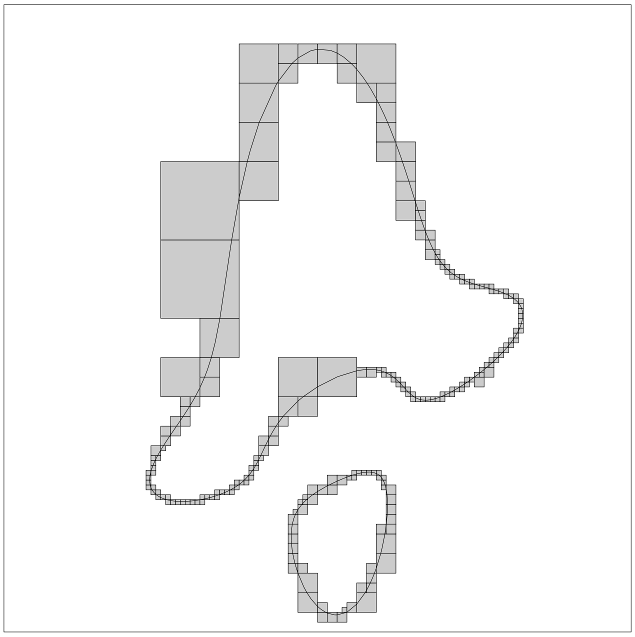
Sextic approximating a Mig outline computed with software by T. Tasdizen at `http://www.lems.brown.edu/~tt/`

$(L = 3.01, \quad \delta = 0.99)$



Pisces logo: `http://www.geom.umn.edu/~fjw/pisces/`

$(L = 2.71, \quad \delta = 0.95)$



Quartic from [34]:

$0.004 + 0.110x - 0.177y - 0.174x^2 + 0.224xy - 0.303y^2 - 0.168x^3 + 0.327x^2y - 0.087xy^2 - 0.013y^3 + 0.235x^4 - 0.667x^3y + 0.745x^2y^2 - 0.029xy^3 + 0.072y^4 = 0$

$(L = 2.19, \quad \delta = 0.99)$

Figure 3: Examples of adaptive approximation (cont.).