Comput. & Graphics Vol. 15, No. 3, pp. 331–340, 1991
Printed in Great Britain.

0097-8493/91 $3.00 + .00

INTERVAL METHODS IN COMPUTER GRAPHICS

KEVIN G. SUFFERN
School of Computing Sciences, University of Technology, Sydney,
P.O. Box 123 Broadway, NSW 2007, Australia

and

EDWARD D. FACKERELL
Department of Applied Mathematics, The University of Sydney, NSW 2006, Australia

Abstract—The methods of interval arithmetic are applied to graphics algorithms for contouring functions of two variables and rendering implicit surfaces. Interval methods result in algorithms that are guaranteed not to miss parts of the contours or surfaces down to a specified size in the viewing region. Thus, they provide a degree of robustness to the algorithms which is difficult to achieve when point sampling alone is used to detect the contours and surfaces.

1. INTRODUCTION

This article discusses techniques for improving the robustness of algorithms designed to plot contours of functions of two variables of the form $z = f(x, y)$, and render implicit surfaces of the form $f(x, y, z) = 0$. Graphics algorithms for performing the above tasks (of which many have been developed) basically involve two stages. The first is that of detecting the contour or surface, and the second is the rendering process. Of the two stages, the first is the most important, because unless the detection is done correctly, the rendering cannot be correct.

The process of *point sampling*, which involves evaluating the signs of the function at the corners of rectangular boxes in the plotting region, is frequently used for detection purposes. If any of the signs are different, the contour or surface passes through the box and is thus detected by the point sampling. The problem with this technique is that point sampling alone cannot *guarantee* the contour or surface will be detected. Fig. 1 shows several cases in two dimensions where a contour segment will be missed if the function is evaluated only at the corners of the squares. The contour segment in (a) crosses the edge of the square twice, with the result that the signs at all four corners are the same. In (b), there are two contour segments present with the result again that all signs are the same. Small closed contours, as in (c), are particularly easy to miss and the only way to detect them is to use a finer subdivision of the plotting region for detection purposes, which can be very inefficient. In addition, long thin contours, as in (d), can still be missed, even though they are physically larger than the squares. In Fig. 1, only the contour section in (e) would be detected. The same problems exist in three dimensions when the point sampling is used to detect sections of surfaces.

There is obviously a need for detection techniques that are guaranteed not to miss sections of the contours or surfaces in the plotting region, but for this to be achieved, the point sampling technique described above needs to be supplemented by auxiliary information about the functions.

The auxiliary information used by Kalra and Barr[3] for ray tracing implicit surfaces was the Lipschitz constant of the functions. The Lipschitz constant is an upper bound on the magnitude of the gradient of the function in a given region of space, and can thus be very difficult to compute for completely arbitrary implicit functions. Consequently, their algorithm is only applicable to functions for which they can compute the Lipschitz constant.

The methods of interval analysis also provide the necessary auxiliary information to provide guaranteed detection algorithms but, in general, are much easier to work with for arbitrary functions than the calculation of Lipschitz constants. Interval techniques have been used in numerical analysis for at least 25 years but have seen very little use in computer graphics. Mudur and Koparkar[7] discussed interval methods for processing geometric objects with some graphics applications for parametric surfaces. Their study also includes an elementary discussion of interval techniques.

Section 2 of this article is an elementary introduction to interval arithmetic, sections 3 and 4 discuss interval algorithms for drawing contours and rendering implicit surfaces, and section 5 discusses directions for future research.

2. INTERVAL ARITHMETIC

Several books have been written on interval analysis, starting with the classic book by Moore[5], who developed most of the original theory of intervals. Two recent books which contain extensive bibliographies are by Ratschek and Rokne[8, 9].

An interval $I = [a, b]$, $a \le b$ is a set of real numbers defined by $[a, b] = \{x | a \le x \le b\}$. If A and B are intervals and $*$ denotes one of the arithmetic operators $+, -, \cdot$ and $/$, then $A * B$ is defined by

$$A * B = \{x * y | x \in A, y \in B\}. \qquad (1)$$

The real number a is considered to be an interval $a = [a, a]$, which means definition (1) gives well-defined meanings to expressions such as aA, $a + A$, A/a,

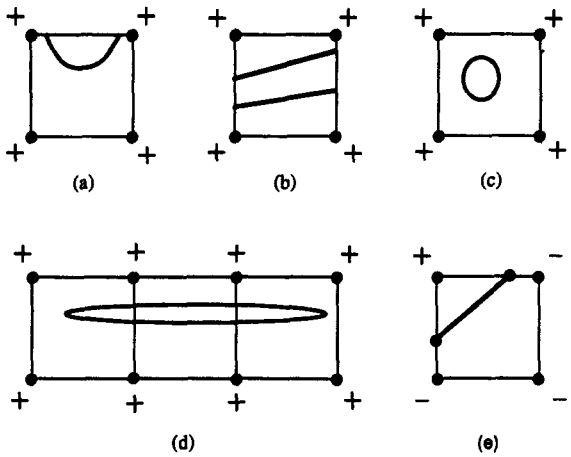

Fig. 1. Contour segments in detection squares. Black circles indicate where the sign of the function is evaluated together with (arbitrary) signs (+ or −), or where the ends of contour segments will be plotted. (a) Highly curved contour segment crosses an edge twice. (b) Two contour segments exist in the square. (c) Small closed contour does not cross any edges. (d) Long thin closed contour exists in several adjacent squares but crosses edges twice. (e) Single contour segment crosses two edges once and is detected.

$(-1)A = -A$. Although definition (1) is of no use for practical calculations, Moore[3] proved that eqn. (1) is equivalent to the following set of constructive rules:

$$
\begin{aligned}
[a, b] + [c, d] &= [a + c, b + d] \\
[a, b] - [c, d] &= [a - d, b - c] \\
[a, b] \cdot [c, d] &= [\min(ac, ad, bc, bd), \\
&\qquad \max(ac, ad, bc, bd)] \\
[a, b]/[c, d] &= [a, b] \cdot [1/d, 1/c] \\
&\qquad \text{provided} \quad 0 \notin [c, d]. \quad (2)
\end{aligned}
$$

The common set theoretic operations, *intersection* and *union*, are applicable to intervals:

$$
\begin{aligned}
A \cap B &= \{x | x \in A \text{ and } x \in B\} \\
A \cup B &= \{x | x \in A \text{ or } x \in B\}.
\end{aligned}
$$

The above definition of the division of intervals needs to be extended to the case where $0 \in [c, d]$, which leads to unbounded intervals. Let $0 \in [c, d]$ and $c < d$, then

$$
\begin{aligned}
&[a, b]/[c, d] \\
&= [b/c, +\infty) \quad \text{if} \quad b \le 0 \quad \text{and} \quad d = 0 \\
&= (-\infty, b/d] \cup [b/c, +\infty) \\
&\qquad \text{if} \quad b \le 0, c < 0, \quad \text{and} \quad d > 0, \\
&= (-\infty, b/d] \quad \text{if} \quad b \le 0 \quad \text{and} \quad c = 0 \\
&= (-\infty, a/c] \quad \text{if} \quad a \ge 0 \quad \text{and} \quad d = 0 \\
&= (-\infty, a/c] \cup [a/d, +\infty) \\
&\qquad \text{if} \quad a > 0, c < 0, \quad \text{and} \quad d > 0 \\
&= (-\infty, +\infty) \quad \text{if} \quad a < 0 \quad \text{and} \quad b > 0. \quad (3)
\end{aligned}
$$

In addition, $[a, b]/0 = (-\infty, +\infty)$. These formulae are from Hansen[2]; notice how the result can be the union of two semi-infinite intervals. The computer implementation of unbounded intervals is discussed by Ratschek and Rokne[9].

We now discuss *rational* functions and their interval extensions. A rational function is a function that can be evaluated using a finite number of the arithmetic operations +, −, ·, and /. Ratios of polynomials are rational functions. Given a rational function $f(x_1, x_2, \cdots x_n)$: $\mathbb{R}^m \to \mathbb{R}$ the *natural interval extension* of f, denoted by $F(X_1, X_2, \cdots X_n)$, is obtained by replacing each occurrence of the x_i's in f by intervals X_i, and evaluating the resulting interval expression using the definitions (2) and (3). The result is an interval, or a union of intervals, not a single real number.

The primary motivation for using interval analysis, not only in computer graphics, but in almost all other applications, is that the interval extension of a function provides bounds for the variation of the function. This comes from the *fundamental property of interval arithmetic*: $x \in X \Rightarrow f(x) \in F(X)$. To see what this means, consider the following example. Let $f(x, y)$ be a function of two variables, and let x and y be confined to the rectangular region $a \le x \le b$, $c \le y \le d$ of the (x, y) plane. If we compute the natural interval extension $F(X, Y)$ of $f(x, y)$, where $X = [a, b]$, and $Y = [c, d]$, the actual variation of the function $f(x, y)$ over the region $a \le x \le b$, $c \le y \le d$ is then contained within the interval $F(X, Y)$. The bounds on the function represented by the ends of the interval $F(X, Y)$ may not be very tight, but the values of the function $f(x, y)$ are *guaranteed* not to lie outside this interval. For graphics applications this leads to guaranteed tests that a contour or surface does *not* lie in a region of space. These tests are discussed in the following two sections.

The fundamental property of interval arithmetic also applies to functions that are not rational, but interval extensions must be written for any functions we wish to investigate. It is simple to write interval extensions for the elementary functions, and of these, functions such as e^x, $\ln x$ and $\sqrt{x}$ are the simplest because of their monotonicity. Thus, if $X = [a, b]$, interval extensions of these functions are

$$
\begin{aligned}
EXPI(X) &= [e^a, e^b] \\
LNI(X) &= [\ln a, \ln b] \\
SQRTI(X) &= [\sqrt{a}, \sqrt{b}]
\end{aligned}
$$

and these give the exact ranges of the functions over the interval X. Other functions such as $\sin(x)$, $\cos(x)$, *etc.*, are not monotonic, and are thus slightly more complicated, but since their critical points are well

```
procedure create_tree ( depth : integer ;
                        x,y,d : real ;
                        function f (x, y : real ;) : real ) ;

    procedure subdivide ;
        begin
            create_tree (depth+1,x,y,d/2,f) ;
            create_tree (depth+1,x+d/2,y,d/2,f) ;
            create_tree (depth+1,x+d/2,y+d/2,d/2,f) ;
            create_tree (depth+1,x,y+d/2,d/2,f)
        end ;

    begin { create_tree }
        if depth < search_depth
            then subdivide
            else if contour_present (x,y,d,f)
                then if depth < plot_depth
                        then subdivide
                        else plot (x,y,d,f)
    end ; { create_tree }
```

Fig. 2. Algorithm 1, for plotting contours (see text for explanation).

known, their interval extensions are usually straightforward to implement.

3. CONTOURING ALGORITHMS

Fig. 2 shows algorithm 1, a simple grid-based algorithm from Suffern[13], for contouring functions of two variables. The procedure *create_tree* is called with *depth* = 0, the parameters x and y are set to the lower left coordinates of the square plotting area, and d to the size of the plotting area. The algorithm uses a quadtree subdivision to uniformly subdivide the plotting area down to *depth* = *search_depth*(*depth* = 0 corresponds to whole plotting area), and then to further subdivide those quadrants that contain the contour down to *depth* = *plot_depth* > *search_depth*. Contour segments are detected in the procedure *contour_present* using point sampling, and plotted by the procedure *plot*(x, y, d, f). Quadrants at *depth* = *plot_depth* are known as *plotting cells*. This algorithm has the advantage over older grid-based algorithms, *i.e.*, [15], in that the plotting area only needs to be finely subdivided in the neighbourhood of the contour. However, the point sampling can miss contour segments as discussed in the introduction. Fig. 3 shows some contours of the simple function

$$f(x, y) = x^2 + y^2 \qquad (4)$$

plotted with this algorithm. Although most of these contours can be successfully detected with low search depths of 1 or 2, the small innermost contour needs a *search_depth* = 7 for detection. This corresponds to dividing the plotting area into the fine search grid shown in Fig. 4, which is very inefficient.

In contrast, interval methods allow this contour to be detected with a *search_depth* of (effectively) zero (see discussion below of algorithm 3). This is because they allow us to find with *absolute certainty*, parts of the plotting region where the contour cannot exist. This is best explained with a diagram. Fig. 5 shows the variation of a function $f(x)$ of one variable over an interval $X = [a, b]$ on the x axis. The diagram also shows the interval extension of the function $F(X) = [A, B]$, and some values c_1, c_2, and c_3 that are to be tested for inclusion in the interval $F(X)$. Since the variation of the function always lies within its interval extension, if a value c_i does not belong to the interval $F(X)$, it

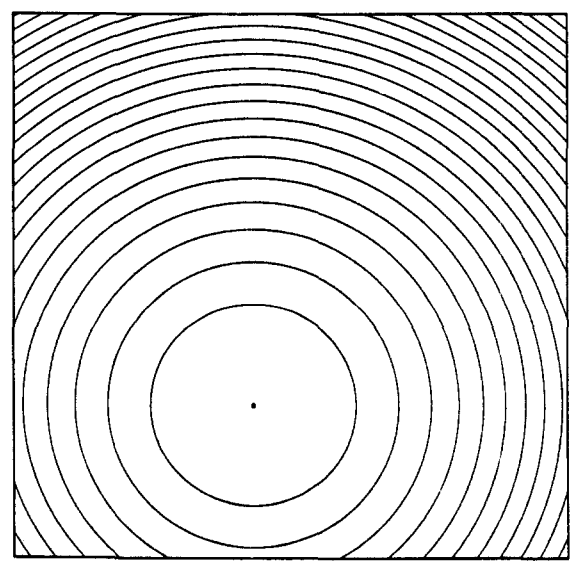

Fig. 3. Contours of the function $f(x, y) = x^2 + y^2$ over the region $-2.6 \le x \le 3.6$, $-1.7 \le y \le 4.7$. The lowest contour is $c = 0.0005$, and the highest is $c = 30.0$. These were plotted by algorithm 1, with *search_depth* = 7 and *plot_depth* = 8.

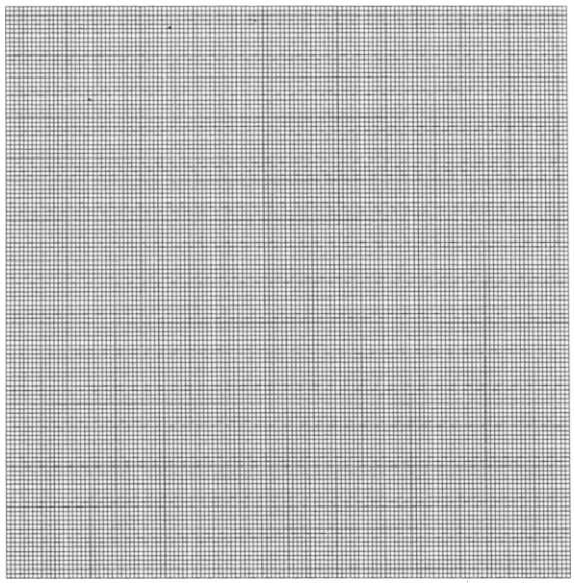

Fig. 4. Search grid corresponding to *search_depth* = 7 used for plotting Fig. 3.

cannot be a value of the function in the interval $X = [a, b]$. This is the case for $c_1 \notin F(X)$. If the value does belong to the interval $F(X)$, it may or may not be a value of the function. The value $c_2 \in F(X)$ is not a function value, but $c_3 \in F(X)$ is. In two dimensions, if a contour level c does not belong to the interval extension $F(X, Y)$ of a function $f(x, y)$ over a rectangular region $x \in X$, $y \in Y$ of the (x, y) plane, the contour *cannot* exist in that region. This is a *contour exclusion test* which allows that region to be excluded from further consideration.

The above discussion suggests the following extremely simple interval-based contouring algorithm, which appears in Fig. 6. In algorithm 2, *depth*, *plot_depth*, x, y, and d have the same use as in algorithm 1, and *FI* is a function which returns the natural interval extension of the function f, which is being contoured. The function *contour_not_present* returns true if the contour level $c \notin FI(XI, YI)$. Any quadrants at *plot_depth* are coloured black by the procedure *colour_quadrant_black*. Algorithm 2 uses only the interval extension of the function, and not the function itself.

Fig. 7 displays some of the contours from Fig. 3 plotted on a 256 × 256 raster window with *plot_depth* = 8. Algorithm 2 is not a practical algorithm because its resolution is limited to the resolution of the raster display, and this limits its usefulness for hard copy. In addition, it is inefficient because high values of *plot_depth* are required to produce thin contours. Even with these high values, the fact that it draws sections of the plotting area where *contour_not_present* returns *false*, means the contours can still be very thick. This is demonstrated in Fig. 8 which displays some contours of the function

$$f(x, y) = \prod_{j=1}^{m} [(x - x_j)^2 + (y - y_j)^2] \qquad (5)$$

plotted with *plot_depth* = 8.

To make algorithm 2 practical, only two changes are required. First, the function being contoured is added as a parameter to *intervals_tree*, and second, the procedure *colour_quadrant_black* is replaced by the procedure *plot* (x, y, d, f), which uses point sampling to detect a contour segment in the plotting cell and then plots it. The method of *false position* is used to calculate the intersections of the contour segment with the edges of the plotting cell. The resulting algorithm 3 appears in Fig. 9. Of course, not all quadrants at *plot_depth* will contain contour segments, but the interval function *contour_not_present* guarantees that no quadrants larger than the plotting cells are incorrectly discarded. Algorithm 3 is a hybrid algorithm which uses the function for the actual plotting and the interval extension to eliminate areas of the plotting region.

Comparing algorithms 1 and 2 reveals that contours are plotted by algorithm 3 with a search depth of effectively zero, because the code in the body of the procedure *intervals_quadtree* in Fig. 9 is equivalent in its results to

```
if depth < 0   {ie, search_depth = 0}
  then subdivide
  else if contour_not_present (x,y,d,FI)
    then {discard current quadrant}
    else if depth < plot_depth
      then subdivide
      else plot (x,y,d,f).
```

Algorithm 3 was used to reproduce the contours in Fig. 3 with *plot_depth* = 8, and the quadrants that were discarded in detecting the innermost contour appear in Fig. 10. This figure should be compared with Fig. 4. Algorithm 3 is much more efficient than algorithm 1 for the function (4), and Table 1 gives the relative timings. The "3.1" entry in the table indicates algorithm 3 ran 3.1 times faster than algorithm 1.

Figure 11 shows contours of the function (5) produced by algorithm 1 with *search_depth* = 8 and *plot_depth* = 9, and algorithm 3 with *plot_depth* = 9 (both algorithms produced identical contours). Without using intervals, algorithm 1 required *search_depth*

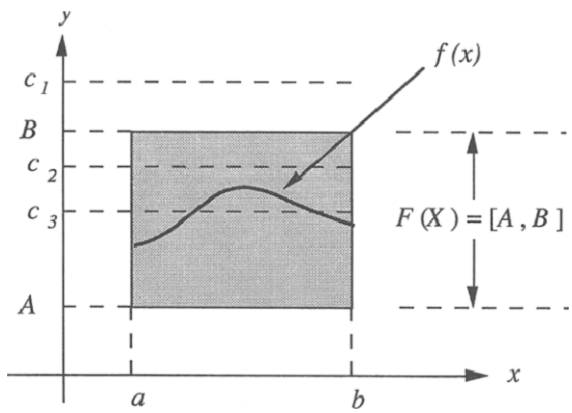

Fig. 5. Range of a one-dimensional function $f(x)$ over a finite interval, and the range of its interval extension $F(X)$.

```
procedure intervals_quadtree ( depth: integer;
                               x, y, d: real;
                               function FI (XI,YI : interval) : interval) ;

  procedure subdivide ;
    begin { subdivide }
      intervals_quadtree (depth + 1, x, y, d / 2, FI) ;
      intervals_quadtree (depth + 1, x, y + d / 2, d / 2, FI) ;
      intervals_quadtree (depth + 1, x + d / 2, y + d / 2, d / 2, FI) ;
      intervals_quadtree (depth + 1, x + d / 2, y, d / 2, FI)

    end ;   { subdivide }

  begin { intervals_quadtree }
    if contour_not_present (x,y,d,FI)
        then { discard quadrant }
        else if depth < plot_depth
                then subdivide
                else colour_quadrant_black (x,y,d)

  end ; { intervals_quadtree }
```

Fig. 6. Algorithm 2 for plotting contours using interval methods.

= 8 to detect the very small closed loops. For the other contours, *search_depth* in the range 5 to 7 was used with *plot_depth* in the range 6 to 8. For this function, algorithm 3 is still faster than algorithm 1, as can be seen by the timings in Table 1, but the speed difference is not great. This is because, in general, it takes longer to evaluate the interval extension of a function than it does to calculate the function itself. The interval operations in eqn. (2) require a number of arithmetic operations for their evaluation, and multiplication and division also require the evaluation of Boolean expressions. For example, the multiplication of two finite intervals requires four multiplications and the extraction of the minimum and maximum values from a list of four values. The complexity of the interval calculations increases again when the infinite interval types of eqn. (3) are included. In addition, the interval arithmetic operations are implemented in our system as function subprograms.

The many long contour segments in Fig. 11 result in many quadrants being checked by *contour_not_present* at high values of *depth* before being discarded. These are clustered along the contours and are shown in Fig. 12 for the $c = 0.1$ contour in Fig. 11. This was plotted with *plot_depth* = 7 in both figures. The relative speed of algorithms 1 and 3 thus depends on the functions being contoured.

The natural interval extensions used in this article

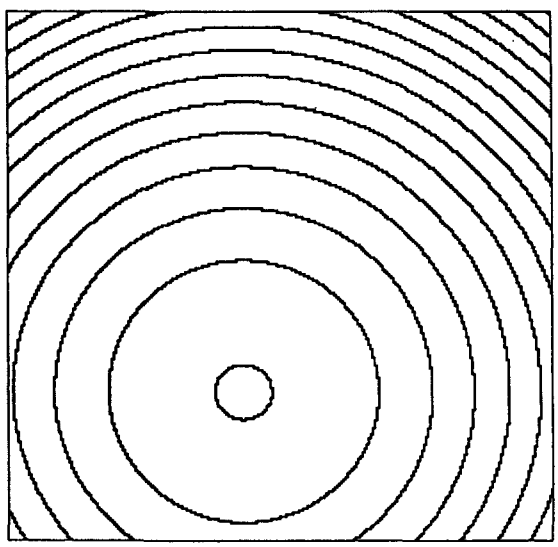

Fig. 7. Some of the contours from Fig. 3 plotted by algorithm 2 with *plot_depth* = 8.

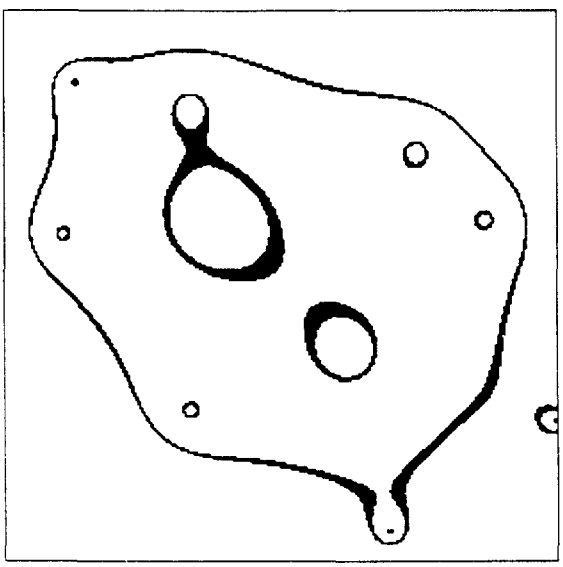

Fig. 8. Some contours of the function (5) plotted by algorithm 2 with *plot_depth* = 8.

```
procedure intervals_quadtree ( depth: integer;
                               x, y, d: real;
                               function f (x,y : real) : real ;
                               function FI (XI,YI : interval) : interval) ;

  procedure subdivide ;
    begin { subdivide }
      intervals_quadtree (depth + 1, x, y, d / 2,f, FI) ;
      intervals_quadtree (depth + 1, x, y + d / 2, d / 2,f, FI) ;
      intervals_quadtree (depth + 1, x + d / 2, y + d / 2, d / 2,f, FI) ;
      intervals_quadtree (depth + 1, x + d / 2, y, d / 2,f, FI)

    end ;   { subdivide }

  begin { intervals_quadtree }
    if contour_not_present (x,y,d,FI)
        then { discard current quadrant }
        else if depth < plot_depth
                then subdivide
                else plot (x,y,d,f)

  end ; { intervals_quadtree }
```

Fig. 9. Algorithm 3 for plotting contours using interval methods.

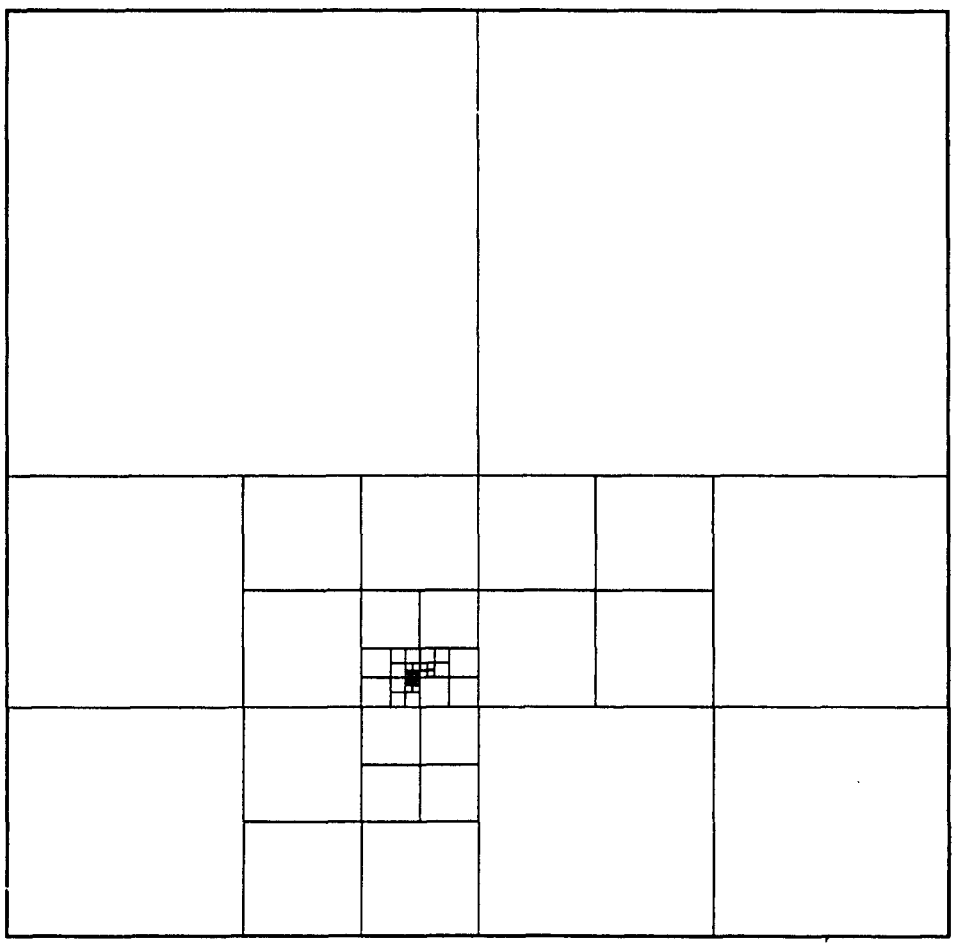

Fig. 10. The squares inside the plotting area (outer square) are quadrants for which *contour_not_present* in algorithm 3 returns *true* when plotting the innermost contour of Fig. 3.

Table 1. Ratio of timings: noninterval methods/ interval methods.

Fig.	Ratio of timings
3	3.1
11	1.1
13	0.8
15	0.5
16	48.2
17	0.9

are the simplest interval representations to calculate, but usually give the widest intervals. Narrower intervals can be constructed, but these usually involve derivatives of the functions. Narrower intervals could make the algorithms more efficient in one way, because quadrants could be discarded at higher depths, but these would require more calculation, thus reducing the efficiency.

4. RENDERING IMPLICIT SURFACES

The techniques discussed for contouring functions of two variables are readily extended for rendering implicit surfaces. Mathematical surfaces can be rendered in many different ways, and here we discuss two different techniques. The first technique, based directly on algorithm 3, allows the user to plot contours of a function $f(x, y, z) = 0$ in a series of planes parallel to the coordinate planes. Since one of x, y, or z is constant in these planes, this technique reduces the problem of rendering a surface to the repeated application of contouring a function of two variables. The user first specifies a cube in which the function is to be rendered (the *viewing cube*), which planes to use, the number of planes, *plot_depth* (as in algorithm 2), and the view point. The contours are then drawn in perspective, with depth cueing used as an option.

Fig. 13 shows contours of a function which is the three-dimensional analogue of the function (5). This figure shows contours on 41 planes parallel to the (x, y) plane. Without using intervals, *search_depth* = 6 and *plot_depth* = 6 were required to detect the small disconnected part of the surface on the bottom right. The interval algorithm was used with *plot_depth* = 6, and Table 1 gives the relative timings. For this function, the interval methods are slower than the noninterval methods.

The second technique uses an octree subdivision of the viewing cube to plot the surface. Algorithm 4, which is a generalisation to three dimensions of algorithm 3, appears in Fig. 14. The output is a polygonisation of the implicit surface in the viewing cube which can be rendered in a variety of ways. The function *surface_not_present* returns *true* if the surface is not present in an octant, and the procedure *plot_3D* calculates the intersections of the surface with the three-dimensional plotting cell edges. Wire frame views can be produced, hidden lines can be removed, or shaded images can be produced. When shaded images are chosen, a single light source is used with constant shading of the polygons. Details are given in Suffern [10–13]. These references also discuss the version of algorithm 4 which does not use intervals. In the noninterval algorithm, the viewing cube is subdivided to *search_depth*, and those octants that contain the surface (as detected by point sampling) are further subdivided to *plot_depth*. Bloomenthal [1] discusses similar algorithms for polygonising implicit surfaces.

Some results are given in Figs. 15–17. Fig. 15 shows the function

$$f(x, y, z) = \sum_{j=1}^{n} \frac{q_j}{(x - x_j)^2 + (y - y_j)^2 + (z - z_j)^2} - c \quad (6)$$

which can consist of an arbitrary number of small separate closed surfaces surrounding the points (x_j, y_j, z_j). Thus, it is a good candidate for testing interval techniques, as the separate sections can easily be missed if intervals are not used. Without using intervals, *search_depth* = 5 was required to detect all 12 separate parts of the surface in the viewing cube. Fig. 16 shows the function

$$f(x, y, z) = x^2(y^2 + z^2 + a)^{2n} + y^2 + z^2 - r^2 \quad (7)$$

where a, r, and n are constants. The surface is plotted by algorithm 4. When intervals are not used, *search_depth* = 6 is required to detect all parts of the surface because with *search_depth* = 5 the point sampling misses the two pointed ends.

Finally, Fig. 17 displays the function

$$f(x, y, z) = (x^2 - yx + z^2y)e^{xyz}. \quad (8)$$

The timing results appear in Table 1, where it can be seen that the relative speeds of the interval and non-

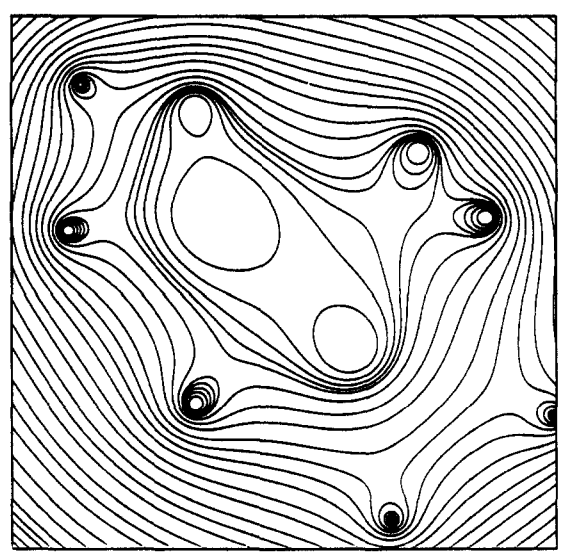

Fig. 11. Contours of the function (5) in the region $-1 \le x$, $y \le 1$ plotted by algorithm 3. The lowest contour is $c = 0.0006$, and the highest is $c = 8000$. The values of *plot_depth* ranged from 9 for $c = 0.0006$ to 6 for $c = 8000$.

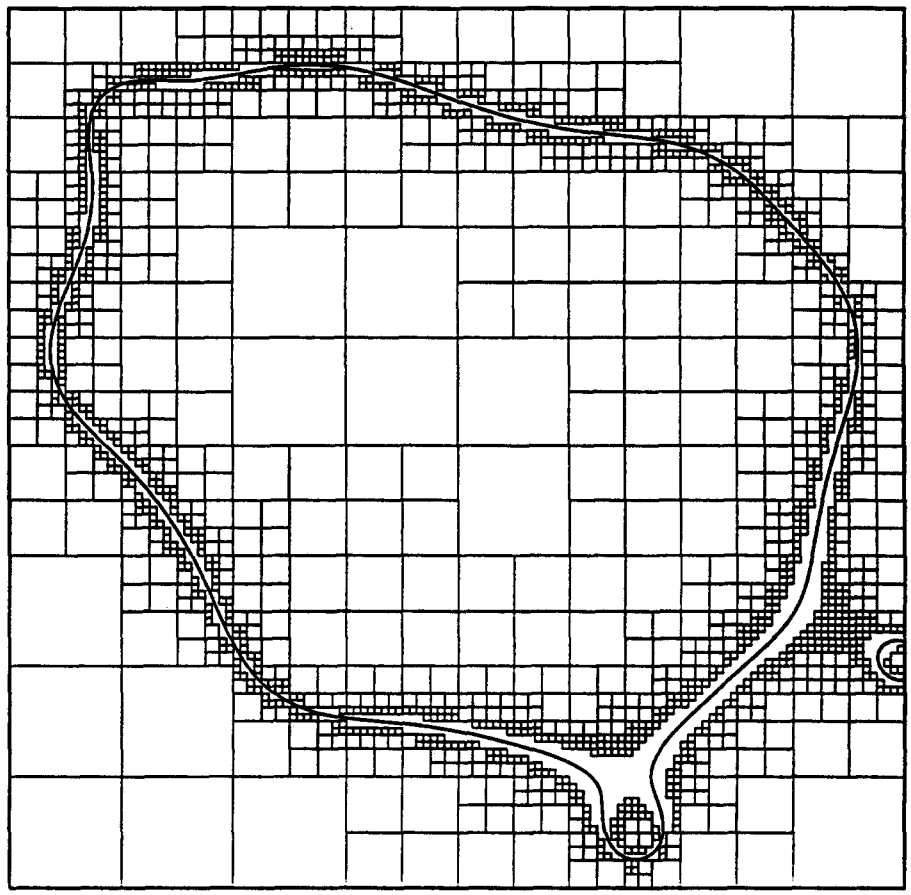

Fig. 12. Quadrants which returned *contour_not_present* = *true* in algorithm 3 for the contour $c = 0.1$ in Fig. 11 with *plot_depth* = 7.

interval algorithms are strongly dependent on the surface. Where the surface is distributed throughout the viewing volume, as in Figs. 13, 15, and 17, the interval methods slow down the algorithms, because high subdivision depths are required before parts of the viewing volume can be discarded. In Fig. 16, the surface occupies only a small percentage of the viewing volume (not drawn in this figure), with the result that large regions of the viewing volume can be discarded at low subdivision depths. The subdivision process here is similar to that in Fig. 10, and the interval algorithm is approximately 48 times faster for this surface.

5. DIRECTIONS FOR FUTURE RESEARCH

There are number of aspects of the algorithms presented here which will benefit from future research. First, the robustness of the algorithms can be further improved by applying interval methods to the plotting cells. This can help guarantee that algorithm 3 will correctly handle the (rare) plotting cells shown in Fig. 1(b) where two contour segments are present. The current procedure *plot* in algorithm 3 would not plot these contours at all. Plotting cells of this type require a *contour inclusion test* so that they can be further subdivided until, ultimately, the only type of cell plotted is similar to Fig. 1(a). Interval methods can be used to test if there are multiple contour segments in a plotting cell, but the intervals used would have to be much tighter than those provided by the natural interval extensions. Virtually every plotting cell we tested with

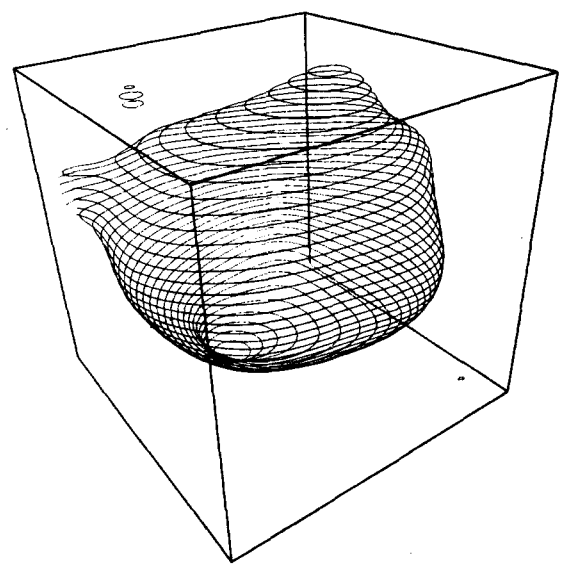

Fig. 13. Perspective view of a function which is the three-dimensional analogue of eq. (5) plotted with *plot_depth* = 6.

```
procedure intervals_octree (depth : integer ;
                            x, y, z, d : real ;
                            function f (x,y,z : real) : real ;
                            function FI (XI,YI,ZI : interval) : interval ) ;

  procedure subdivide ;
    var
      j, k, l : integer ;
    begin  { subdivide }
      for j := 0 to 1 do
        for k := 0 to 1 do
          for l := 0 to 1 do
            intervals_octree (depth + 1,
                              x+j*d/2, y+k*d/2, z+l*d/2, d/2,f,FI) ;
    end ;   { subdivide }

  begin   { intervals_octree }
    if surface_not_present (x, y, z, size, FI)
      then { discard current octant }
      else if depth < plot_depth
             then subdivide
             else plot_3D (x, y, z, d,f);
  end ;      { intervals_octree }
```

Fig. 14. Algorithm 4 for rendering implicit surfaces using interval methods.

contour_not_present and *surface_not_present* returned *false* because of the broadness of the intervals and the proximity of the plotting cells to the contours or surfaces. Saddle points (where contours cross) can be handled in a similar manner, as can the analogous cases for implicit surfaces.

Second, the algorithms described here use a fixed, user-specified *plot_depth*, regardless of the local curvature of the contours or surfaces. Algorithms which employ plot depths which depend on the local curvature can be much more efficient, and these techniques need to be incorporated into the current algorithms. Curvature-driven plot depths were used for contouring by Suffern[10] and polygonising implicit surfaces by Bloomenthal[1]. Automatic algorithms which require a minimum of user-specified input parameters are desirable, and curvature-based interval algorithms similar to algorithms 3 and 4 would, in principle, need no parameters other than those to specify the viewing geometry.

6. CONCLUSIONS

Interval methods, which are widely used in numerical analysis, have important applications in computer

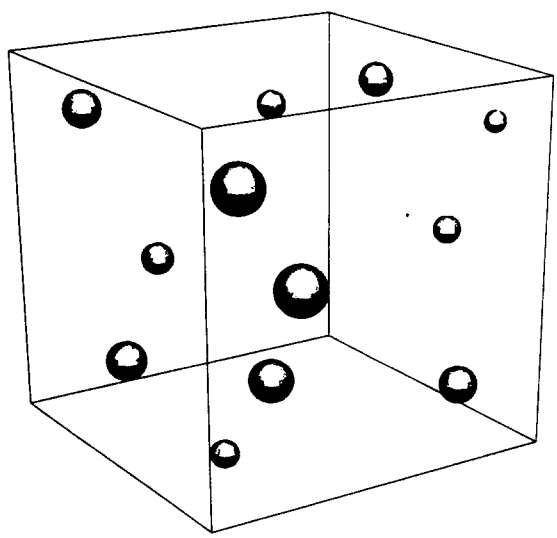

Fig. 15. Shaded perspective image of the surface defined by eq. (6) plotted by algorithm 4 with *plot_depth* = 6.

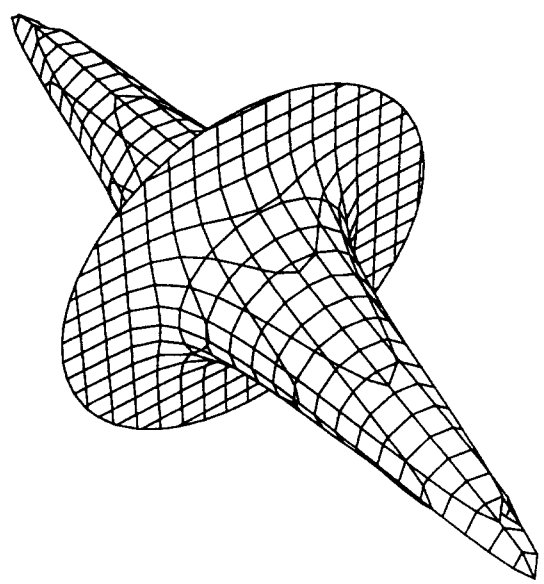

Fig. 16. Perspective view of the surface defined by eq. (7) with $a = 0.985$, $r = 0.5$, and $n = 50$, plotted by algorithm 4 with *plot_depth* = 6.

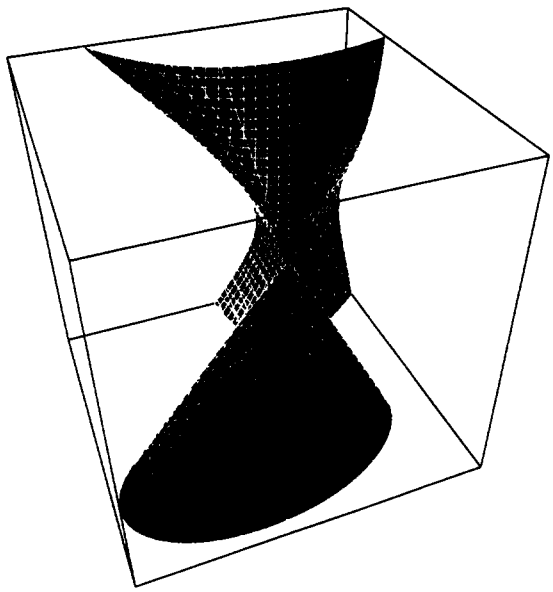

Fig. 17. Shaded image of the surface defined by eq. (8) plotted by algorithm 4 with *plot_depth* = 6.

graphics. They can be used to improve the robustness of algorithms for contouring functions of two variables, and rendering implicit surfaces. In algorithms using space subdivision techniques they can *guarantee* that no part of the viewing region above a user-specified size is discarded that could *possibly* contain part of the contour or surface. This is something that point-sampling techniques alone are unable to do. Because interval arithmetic is slower than real arithmetic, interval algorithms often run slower than their noninterval counterparts, but can also run considerably faster. Relative speeds depend on the type of contours or surfaces being rendered. Depending on the type of graphics application, robustness can be far more important than speed. In principle, algorithms which do not use interval methods can achieve the same degree of robustness as the interval algorithms presented here, but only at the cost of *always* uniformly subdividing the viewing region into a very fine grid to detect the contour or surface. Interval methods achieve the same robustness without the fine subdivision.

The algorithms discussed here produce polygonisations of implicit surfaces, but interval methods can also be used to produce robust algorithms for ray tracing implicit surfaces in a manner similar to the Lipschitz constants used by Kalra and Barr[3]; see also Mitchell[4].

REFERENCES

1. J. Bloomenthal, Polygonization of implicit surfaces. *Comp. Aided Geom. Design* **5**, 341–355 (1988).
2. E. R. Hansen, Global optimization using interval analysis—the multidimensional case. *Numerische Mathematik* **34**, 247–270 (1980).
3. D. Kalra & A. H. Barr, Guaranteed ray intersections with implicit surfaces. *Comp. Graphics* **23**, 297–306 (1988).
4. D. Mitchell, *Graphics Interface '90*, 68–72 (1990).
5. R. E. Moore, Interval arithmetic and automatic error analysis in digital computation. PhD Thesis, Stanford University (1962).
6. R. E. Moore, *Interval Analysis*, Prentice-Hall, Englewood Cliffs, NJ (1966).
7. S. P. Mudur & P. A. Koparkar, Interval methods for processing geometrical objects. *IEEE Comp. Graphics and Applications* **4**, 7–17 (1984).
8. H. Ratschek & J. Rokne, *Computer Methods for the Range of Functions*, Ellis Horwood, London (1984).
9. H. Ratschek & J. Rokne, *New Computer Methods for Global Optimization*, Ellis Horwood, London (1988).
10. K. G. Suffern, Quadtree algorithms for contouring functions of two variables. School of Computing Sciences, University of Technology, Sydney, *Technical Report* 87.1 (1987).
11. K. G. Suffern, An octree algorithm for displaying implicitly defined mathematical functions, School of Computing Sciences, University of Technology, Sydney, *Technical Report* 87.9 (1987).
12. K. G. Suffern, Recursive space subdivision techniques for displaying implicitly defined surfaces, *Ausgraph* '89 Proceedings, 239–249 (1989).
13. K. G. Suffern, Quadtree algorithms for contouring functions of two variables. *The Comput. J.* **33**, 402–407 (1990).
14. K. G. Suffern, An octree algorithm for displaying implicitly defined mathematical functions. *The Australian Comput. J.* **22**, 2–10 (1990).
15. D. C. Sutcliffe, An algorithm for drawing the curve $f(x, y) = 0$. *The Comp. J.* **19**, 246–249 (1976).