Quadtree Algorithms for Contouring Functions of Two Variables

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

Two grid-based algorithms are presented for contouring functions of two variables over square plotting areas. The algorithms are simple to code, and achieve considerable efficiency through the use of quadtree techniques to produce non-uniform subdivisions of the plotting areas. One algorithm uses curvature properties of the functions being plotted to produce plotting cells of optimal size, and their relative speeds depend on the complexity of the functions. Both algorithms can be adapted for plotting over rectangular areas.

Received February 1988, revised April 1989

1. INTRODUCTION

Methods for contouring functions of two variables are of two general types: grid methods and contour-following methods. Grid methods are the most widely used because of their simplicity,[1-2], and work by defining a grid over the plotting area and then plotting the contours in each cell in the grid. It is most common to use a grid that is fine enough to allow a linear approximation to the function to be used over the individual cells, although quadratic approximations can also be used.[3] A disadvantage of most grid methods is that they require very fine grids to cover areas where the contours have high curvature.

Contour-following methods use one or more points on a contour (usually located by using a coarse grid) to predict where another point on the contour lies.[4-6] They may require the first and second partial derivatives of the function to be calculated at each step,[5-6] and can be complicated to code compared with grid methods. They do, however, have the advantage of providing a plotting step size which varies continuously with the curvature of the contours.

We present here two algorithms that have the advantage of grid methods in being simple to code, but do not share their disadvantage of using a uniformly fine grid to plot highly curved contours. The algorithms are grid methods that use quadtrees to provide an adaptive subdivision of the plotting area. This produces grids which are either fine only in the neighbourhood of a contour, or whose size varies with the curvature of the contour. Before discussing the algorithms themselves, Section 2 discusses some of the plotting details that occur within the individual grid cells.

2. PLOTTING DETAILS

The functions we wish to contour are of the form $z = f(x, y)$, and the algorithms plot a series of level contours where

$$f(x, y) - c = 0 \qquad (1)$$

and c is the contour level. Plotting is done in the individual cells (or quadrants) created by the quadtrees, and point sampling is used to detect the presence of a contour in a cell. The function (1) is evaluated at the four corners of a cell, and unless all four values have the same sign a contour segment is present. A danger with all point-sampling techniques is that contour segments can be missed. For example, this technique can miss a contour that intersects one or more sides of a cell twice, and will miss small closed contours that do not intersect the cell sides. When plotting arbitrary functions it is difficult to guarantee that all contour segments will be detected. Exceptions are polynomial functions, where Descartes' rule of signs can be used to detect the existance of a contour no matter what the signs are at the cell corners.

In practice, the cell sizes should be chosen to ensure, whenever possible, that:

(1) all contours or contour segments above a certain size (physical extent in the plotting area) are detected;

(2) with one exception (discussed below), at most a single contour segment exists in each cell;

(3) the contours may be approximated within the cells for plotting purposes by straight-line segments.

There are no guarantees that these conditions will always be satisfied. Both algorithms produce two grids: one for searching for the contours and another for plotting them. The size of the search grid is set by the user to try to satisfy conditions (1) and (2) above. Since the size of the plotting grid will usually satisfy condition (3) (by user specification or program control), the algorithms are required to plot a single straight-line segment in each plotting cell. The intersection of the contour and a plotting cell edge is found by the method of false position.

If the plotting grid is not fine enough, degenerate cells can occur in which the contour intersects each side of the cell. These are shown in Fig. 1 (*a*), and grid methods must decide which of the two configurations shown is correct. This is normally done by further subdividing the cell, but degenerate cells can alternatively be used to plot saddle points. Since contour lines cross in this case, degenerate cells will occur no matter how fine the grid is. The algorithms discussed here treat degenerate cells as saddle points as shown in Fig. 1 (*b*). This allows the plotting of all saddlepoints except those whose position and orientation are such that two contour lines cross the same side of the plotting cell, or skewed configurations, as shown in Fig. 1 (*c*). These would be either drawn incorrectly or missed entirely by the point sampling.

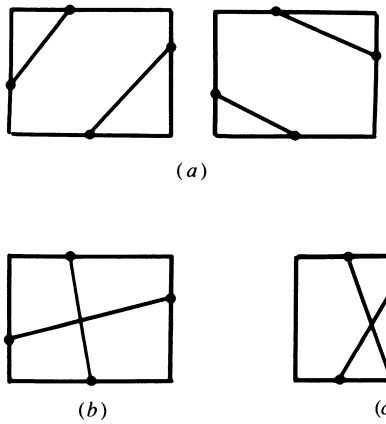

Figure 1. Plotting cells. (*a*) The two possible configurations where the contour crosses each of the four sides of a plotting cell. (*b*) A saddle point. (*c*) A skewed saddle point that the point sampling would miss.

3. QUADTREE SUBDIVISION ALGORITHMS

Quadtrees have many uses in computer science and have found a number of applications in computer graphics and image processing. Samet[7] provides a comprehensive survey of the types and uses of quadtrees. Quadtrees are often used as data structures, but the algorithms here use them only for the adaptive and recursive subdivision of the plotting area.

Before discussing the quadtree algorithms, we discuss some of the disadvantages of traditional grid methods. Fig. 2 shows some contours of $x^2+y^2-c=0$ in the range $c = 0.1(0.1)7.6$ produced by a traditional grid algorithm, and this figure highlights a major disadvantage of these methods. Although the outer contours are drawn reasonably well, the inner contours (which should all be circles), are drawn extremely poorly because here the plotting area was subdivided into a 16×16 grid for plotting purposes. This is adequate for the outer contours but is much too coarse for the inner contours, and only samples the innermost contour four times, resulting in a gross violation of condition (3).

If a finer 64×64 grid is used, the inner circles are drawn well (see Fig. 3), but too many steps are used for the outer circles. However, this is not the worst feature of the traditional algorithm. The worst feature is that for each contour the same uniformly fine grid is used throughout the plotting area. As a consequence, each contour exists in only a small minority of cells, which results in much wasted computation checking function values. The algorithm can be improved by restricting the number of empty cells that must be checked.

Algorithm 1, which appears in Fig. 4 as a Pascal procedure, contains such an improvement. The input parameter *depth* corresponds to the depth in the tree, with the root node *depth* = 0 corresponding to the entire plotting area. The other input parameters are the lower left-hand coordinates x and y of the plotting area (which is square), and its size d. The quantities *search_depth* and *plot_depth* are user supplied, and subdivision is automatically carried down to the level *search_depth*, without checking if a contour is present in any quadrant. The user decides how fine a grid is necessary to search for contours and sets *search_depth* accordingly. Once

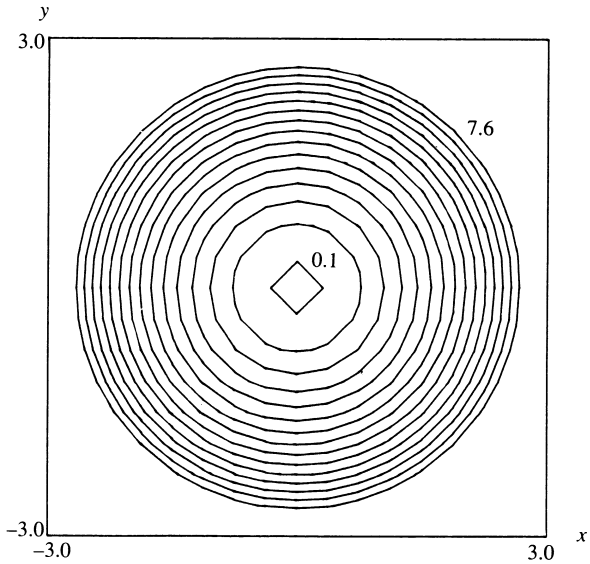

Figure 2. Contours of $f(x,y) = x^2+y^2-c = 0$ with $c = 0.1(0.1)7.6$ plotted with a traditional grid algorithm using a 16×16 grid.

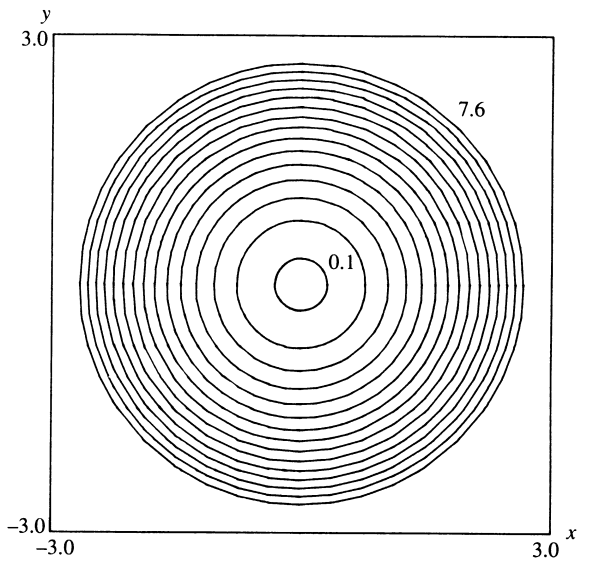

Figure 3. Contours of $f(x,y) = x^2+y^2-c = 0$ with c in the range $c = 0.1(0.1)7.6$ plotted with a traditional grid algorithm using a 64×64 grid.

search_depth is reached, the function *contour_present* point samples the function to check the presence of a contour in the quadrants. Any quadrant containing a contour segment is further subdivided down to *plot_depth*, where *plot_depth* > *search_depth*. The user sets *plot_depth* to give smooth contours according to condition (3) of Section 2, but here the fine subdivision of the plotting area only occurs in the neighbourhood of the contour. The procedure *contour_present* does the point sampling and the procedure *plot* performs the plotting as described in the previous section.

Fig. 3 was reproduced by Algorithm 1 with *search_depth* = 2 and *plot_depth* = 6, and Fig. 5 shows the grids used for the $c = 0.1$ and $c = 7.6$ contours. The dashed lines are the search grid, which is coarser than would be normally used in practice, but works in this case because all the contours are centred on the origin.

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

  procedure subdivide ;
      begin
        create_tree (depth+1,x,y,d/2) ;
        create_tree (depth+1,x+d/2,y,d/2) ;
        create_tree (depth+1,x+d/2,y+d/2,d/2) ;
        create_tree (depth+1,x,y+d/2,d/2)
      end;
  begin {create_tree}
      if depth < search_depth
        then subdivide
        else if contour_present (x,y,d)
                then if depth < plot_depth
                       then subdivide
                       else plot (x,y,d)

end ; {create_tree}
```

Figure 4. Algorithm 1 as discussed in the text.

The solid squares are the plotting cells, each of which contains a contour segment. These are the same size as the cells used for Fig. 3, but now only cover a small fraction of the plotting area.

Algorithm 1 still shares one of the disadvantages of traditional grid methods. The plot depth is the same for all contours, and so again if *plot_depth* is large enough to draw highly curved contours (the inner circles in this case), too many steps will be used to draw contours with lower curvature (the outer circles). Algorithm 2 overcomes this problem and appears in Fig. 6. Here the plot depth is no longer a user-supplied constant, but at each level below *search_level* the function *radius* decides whether to plot the contour or further subdivide the quadrant. This function calculates the radius of curvature R of $f(x,y)$ in the (x,y) plane in each quadrant that contains a contour segment. The expression is

$$R = (f_x^2+f_y^2)^{\frac{3}{2}}/(f_y^2 f_{xx} - 2f_x f_y f_{xy} + f_x^2 f_{yy}) \qquad (2)$$

which is evaluated at the centre of the quadrant and then compared with the quadrant size. Subdivision stops when R is greater than some user-specified multiple n of the quadrant size d. The level of subdivision thus depends on the local curvature of the function, and automatically provides plotting step lengths that are inversely proportional to the radius of curvature of the contours. Time is required to evaluate the derivatives in expression (2), but this is more than offset by being able to use coarser plotting grids (see timings in following section).

The contours of Fig. 3 were repeated with Algorithm 2 using *search_depth* = 2 and $n = 5$. As the contours are very similar in appearance to Fig. 3 there is no point in repeating them here, but Fig. 7 shows the search grid, the plotting cells for the inner and outer contours, and these two contours. The steps used for the outer contour are much larger than those used in Fig. 3, but the steps used for the inner contour are smaller (see Fig. 5). This illustrates the following property of these algorithms: the plotting cell sizes and consequently the step sizes are quantised as $size/2^m$, $m \geqslant 0$. The quadtree algorithms thus do not produce the continuously varying step sizes that some of the contour-following algorithms do. The parameters used in Fig. 7 resulted in one further

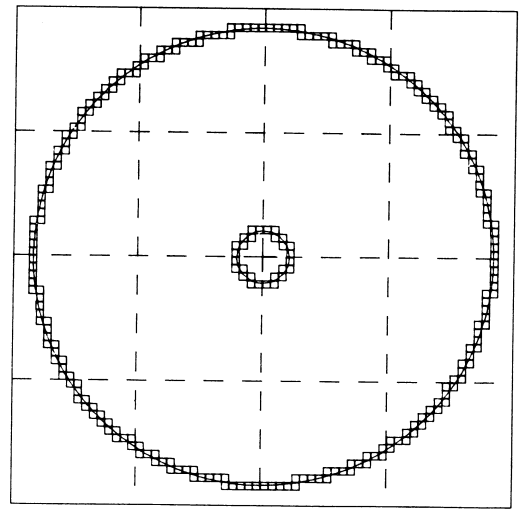

Figure 5. The contours $c = 0.1$ and $c = 7.6$ from Fig. 3 drawn by Algorithm 1, with *search_depth* = 2 and *plot_depth* = 6, together with the search grids (dashed lines) and the plotting cells (small solid squares).

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

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

  begin {create_tree}
      if depth < search_depth
        then subdivide
        else if contour_present (x,y,d)
                then if radius (x,y,d) < n*d
                       then subdivide
                       else plot (x,y,d)

end ; {create_tree}
```

Figure 6. Algorithm 2 as discussed in the text.

subdivision being made when plotting the $c = 0.1$ contour than was used in Figs 3 and 5.

Fig. 8 shows some contours of $\cos(x)\sin(y) - c$ plotted with Algorithm 2 for contours in the range $c = -0.9$ (0.1) 0.9. The parameters used were *search_depth* = 4 and $n = 6$, and this figure shows several saddle points on the $c = 0$ contour. In order to plot this contour correctly, the plotting area was set to $-4.000 \leqslant x \leqslant 4.001$, $-4.000 \leqslant y \leqslant 4.001$. This makes no difference to the appearance of the contours, but prevents the $c = 0$ contour lying exactly along a quadrant edge. This situation could be handled with additional logic in the *contour_present* and *plot* functions.

A question that naturally arises in this context is the following. Is it possible to use curvature information to vary the search depth with the curvature of the function? Such an algorithm would have the advantage of using a fine search grid only in regions of high curvature. To achieve this one may be tempted to replace the expression

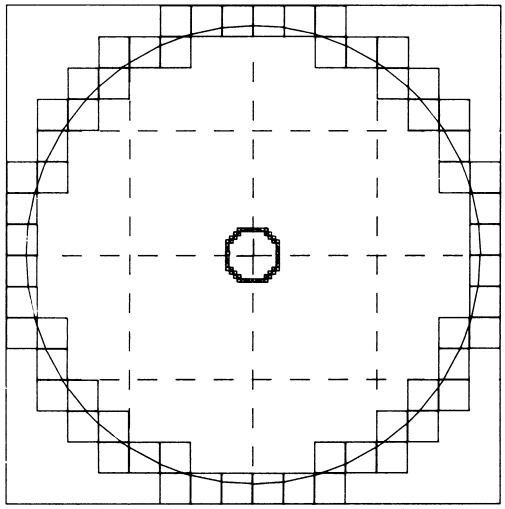

Figure 7. The contours $c = 0.1$ and $c = 7.6$ of $x^2+y^2-c = 0$ produced by Algorithm 2 with *search_depth* = 2 and $n = 5$, together with the plotting cells.

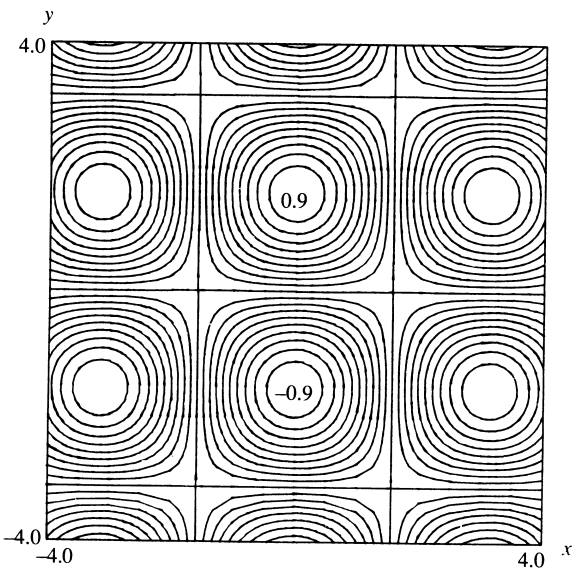

Figure 8. Contours of $\cos(x)\sin(x)-c = 0$ with c in the range $c = -0.9(0.1)0.9$ as plotted by Algorithm 2 with *search_depth* = 4 and $n = 6$.

depth < *search_depth* in Algorithm 2 with *radius* $(x, y, d) < m*d$ where m is a user-supplied constant satisfying $m < n$. This results in a very unrobust algorithm as can be seen by considering x^2+y^2-c again, for which $R = (x^2+y^2)^{\frac{1}{2}}$. Since the procedure *radius* evaluates R at the centre of a quadrant, *radius* will return $R = \sqrt{2}d$ for any quadrant that has a corner at the origin. Consequently *radius* $(x, y, d) < m*d$ will always be true when $m > \sqrt{2}$, leading to infinite recursion and stack overflow. This technique has been tried with other functions too, usually with unpredictable results, and so Algorithm 2 is safer with its user-supplied constant *search_depth*. Further research may reveal a robust method for automatically varying the search depth.

4. TIMING TESTS

To asses the relative speed of the algorithms, several timing tests were performed, and the results appear in Table 1. Three algorithms were compared: Algorithms 1 and 2 and Algorithm 0, which is a traditional grid method. The code for this is not produced here because it is simply a set of nested *for* loops, which call a version of *plot* (x, y, d) modified to do the point sampling, as well as the plotting.

Table 1. Results of timing tests for the indicated algorithms and figures

Algorithm	Fig. 3	Fig. 8	Fig. 9
0	10	61	—
1	4.5	16.3	33
2	2.4	14.8	18

Times are those required to perform the calculations only, and all times are in seconds.

The tests were performed on a Prime 750, and Table 1 lists the figures that were timed. Compared with Algorithm 0, Algorithm 1 is approximately twice as fast for Fig. 3 and approximately four times as fast for Fig. 8. The difference in relative speeds probably arises from the use of recursion in Algorithm 1, which extracts a time penalty. This time penalty is more noticeable in Fig. 3, where the function evaluations take much less time than in Fig. 8. The last row shows the results for Algorithm 2, where for Fig. 3 it is approximately twice as fast as Algorithm 1, but for Fig. 8 the increase in speed is only 10%. In Fig. 8 the time taken for the derivative calculations of $\cos x \sin y$ reduce the relative efficiency of Algorithm 2.

As a final example, Algorithms 1 and 2 were used to draw some contours of the incomplete gamma function

$$\gamma^*(a, x) = \frac{x^{-a}}{\Gamma(a)}\int_0^x e^{-t+a-1}dt$$

To evaluate this we used the series[8]

$$\gamma^*(a, x) = \frac{1}{\Gamma(a)}\sum_{n=0}^{\infty}\frac{(-x)^n}{(a+n)n!}, \quad |x| < \infty$$

with a relative error of 10^{-5}. The factor $1/\Gamma(a)$ was approximated by the first 15 terms of the polynomial expansion given on page 256 of Abramowitz and Stegun,[8] using the first seven significant digits, as all calculations were carried out in single precision. This gave sufficient accuracy for plotting purposes provided $a > -2$. The plotting area had to be chosen so that the subdivision would avoid the evaluation of $\gamma^*(a, x)$ at negative integer values of a. This was done by setting the plotting area to

$$-4.0 \leqslant x \leqslant 1.0, \quad -1.000 \leqslant a \leqslant 2.001$$

Algorithm 1 was used with *search_depth* = 4 and *plot_depth* = 6, Algorithm 2 was used with *search_depth* = 4 and $n = 10$, and Fig. 9 shows the contours from Algorithm 2. As Table 1 shows, Algorithm 2 is almost twice as fast as Algorithm 1. The explanation is that these contours are not highly curved, resulting in less subdivisions in Algorithm 2. The relative speed of Algorithms 1 and 2 is thus highly dependent on the nature of the contours being plotted.

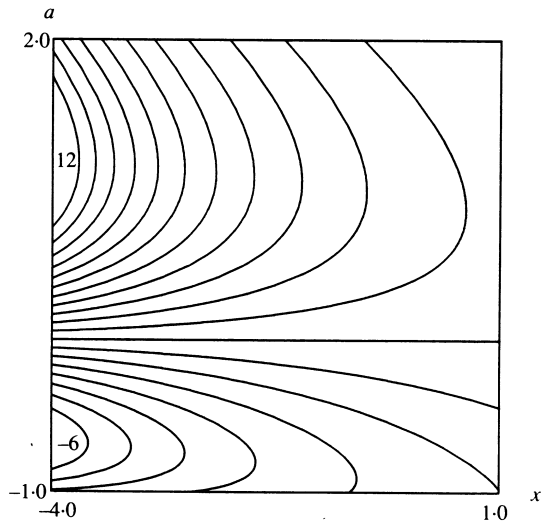

Figure 9. Contours of the incomplete gamma function $\gamma^*(a,x)$ in the range $-6.0(1.0)12.0$ as plotted by Algorithm 2 with *search_depth* = 4 and $n = 10$.

However, plotting $\gamma^*(a,x)$ highlights the major disadvantage of Algorithm 2 from the user's point of view: namely the effort that may be involved to calculate the partial derivatives of the function being plotted. This was trivial for $f(x,y) = x^2+y^2-c$ and $f(x,y) = \cos(x)\sin(x) - c$, but involved some effort for $\gamma^*(a,x)$ where code had to be written to evaluate nine separate series. The result is a faster algorithm, and contours with adaptive step lengths, but this hardly compensates for the extra programming effort involved.

5. RECTANGULAR PLOTS

Both algorithms can be adapted to plot over rectangular regions of the (x,y) plane, and Fig. 10 displays Algorithm 3, a version of Algorithm 1 adapted to do this. Here dx and dy are the extents of the plotting area in the x and y directions respectively.

Fig. 11 shows some contours of the function

$$\cos(x+y)+\sin(x+y)/[(x-2)^2+(y-1)^2]-c=0$$

plotted with *search_depth* = 4 and *plot_depth* = 6. Since each plotting cell in this algorithm has the same aspect ratio as the plotting area, plotting areas with aspect ratios significantly different from unity should be avoided. They can lead to unsatisfactory plots with significantly longer steps in the direction of longest dimension. The aspect ratio of Fig. 10 is 1.25, and for plots with aspect ratios larger than about 1.33, separate plots side by side would be better.

6. CONCLUDING REMARKS

This paper presents some grid-based algorithms which satisfy the following criteria for contouring functions of two variables: they are as simple to code as existing grid methods, and are more efficient.

Algorithms 1 and 2 in Figs 4 and 6 satisfy these criteria. The code for subdividing the plotting area is extremely simple, and the code for carrying out the plotting is no more complicated than for existing grid methods. The timing results in Table 1 show these

```
procedure create_tree (depth      : integer ;
                       x,y,dx,dy  : real    ) ;

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

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

Figure 10. Algorithm 3: a version of Algorithm 1 for plotting over rectangular areas of the (x,y) plane.

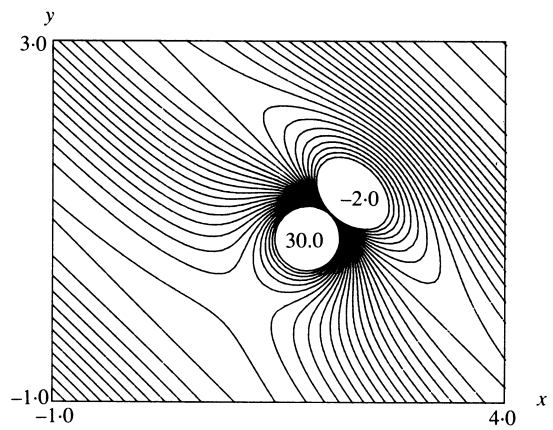

Figure 11. Contours of $\cos(x+y)+\sin(x+y)/[(x-2)^2+(y-1)^2] - c = 0$ for c in the range $c = -2(0.125)30$ as plotted by Algorithm 3 with *search_depth* = 4 and *plot_depth* = 6.

algorithms to be significantly more efficient than traditional grid methods. This efficiency is achieved by using quadtrees to provide a non-uniform subdivision of the plotting area. Which algorithm is best? This depends on the functions being plotted. Algorithm 2 produces contour segments whose lengths are adapted to the local curvature and is faster than Algorithm 1 for all functions that have been tested. However, it may involve considerable programming effort to evaluate the partial derivatives of the function. This must be performed for each function to be plotted and can be a significant overhead for complicated functions, particularly when numerical techniques must be employed. However, this overhead could be avoided by using a computer algebra system as a preprocessor to calculate the partial derivatives.

Both algorithms evaluate the function values a number of times at the same (x,y) values, and for functions that take a significant time to evaluate their efficiency could be further improved by maintaining information about where the function has been evaluated, and its values. This information could be used to avoid the multiple evaluations, but at the cost of considerably increasing the complexity of the code.

The algorithms are recursive, and since recursion is

not a required control structure,[9] they could be coded iteratively. This would have the advantage of enabling them to be written in languages such as FORTRAN 77 or BASIC, but the code would lose the simplicity and elegance of the recursive versions.

Both algorithms can be adapted for plotting over rectangular areas, as discussed in Section 5.

Acknowledgements

This work was carried out while I was on leave at the Center for Interactive Computer Graphics, Rensselaer Polytechnic Institute. I thank Professor Michael Wozney for his hospitality and for providing the facilities which allowed this work to be performed. I also thank Sharon Bohmer for her help in preparing the figures, and Marie Thill (University of Technology, Sydney) for typing the original version of the manuscript.

REFERENCES

1. G. Cottafava and G. Le Moli, Automatic contour map. *Communications of the ACM* **12**, 386–391 (1961).
2. D. C. Sutcliffe, An algorithm for drawing the curve $f(x, y) = 0$. *The Computer Journal* **19**, 216–249 (1976).
3. R. Sibson and G. D. Thompson, A seamed quadratic element for contouring. *The Computer Journal* **24**, 378–382 (1981).
4. K. J. Falconer, *A General Purpose Algorithm for Contouring over Scattered Data Points*. Technical Report, National Physical Laboratory, Teddington, Middlesex, England (1971).
5. I. P. Schagen, Automatic contouring from scattered data points. *The Computer Journal* **25**, 7–11 (1982).
6. K. G. Suffern, Contouring functions of two variables. *Australian Computer Journal* **16**, 102–106 (1984).
7. H. Samet, The Quadtree and related Hierarchical data structures. *ACM Computing Surveys* **16**, 187–260 (1981).
8. M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*. Dover, New York (1975).
9. C. Böhm and A. Jacopini, Flow diagrams, Turing Machines, and languages with only two formation rules. *Communications of the ACM* **9**, 366–371 (1966).