Reliable Computing, , 1–12 ()

Interval Constraint Plotting for Interactive Visual Exploration of Implicitly Defined Relations

TIMOTHY J. HICKEY tim@cs.brandeis.edu
Michtom School of Computer Science, Brandeis University
Waltham, MA 02254

ZHE QIU zqiu@csr.uvic.ca

MAARTEN H. VAN EMDEN vanemden@csr.uvic.ca
Department of Computer Science, University of Victoria
Victoria, B.C., Canada V8W 3P6

Abstract. Conventional plotting programs adopt techniques such as adaptive sampling to approximate, but not to guarantee, correctness and completeness in graphing functions. Moreover, implicitly defined mathematical relations can impose an even greater challenge as they either cannot be plotted directly, or otherwise are likely to be misrepresented. In this paper, we address these problems by investigating *interval constraint plotting* as an alternative approach that plots a *hull* of the specified curve. We present some empirical evidence that this hull property can be achieved by a $\mathcal{O}(n)$ algorithm. Practical experience shows that the hull obtained is the narrowest possible whenever the precision of the underlying floating-point arithmetic is adequate. We describe *IASolver*, a Java applet [9], that serves as testbed for this idea.

Keywords: interval constraints, constraint propagation, interval arithmetic, implicitly defined relations, honest plotting, interactive plotting

1. Introduction

Mathematical modelling in science and engineering often requires

- solving a system of numerical equalities or inequalities. As "equation" is not a sufficiently general term, we refer to them collectively as *constraints*.
- visualizing the solution to the system as a two-dimensional graph. The solution is in general a relation between the two variables selected as coordinates. This relation may be functional or not.

Existing software concentrates on the case where the relation is the functional one specified by $y = f(x)$. The simplest approach is to sample f at equidistant values of x and to connect the resulting points by straight lines. This has the disadvantage that detail is lost where the function changes rapidly. *Adaptive plotting* attempts to allocate the available number of points in such a way that this problem is avoided as much as is possible with the given number of points.

Even when adaptive sampling is used, significant features may fall between successive points and are missed in the plot. Independently of the use of adaptive sampling, two difficulties remain. First, as discussed in [14], numerical rounding errors may accumulate unexpectedly when evaluating complex equations, and thus cause the plot to stray from the correct path. Second, and more seriously, implicitly defined curves, such as $x^2 + y^2 = 1$ and $\sin(x \cos y) = \cos(y \sin x)$ need to be converted to the functional form $y = f(x)$. Sometimes this conversion is straightforward, sometimes not. Without conversion to functional form, one must resort to local methods such as [3, 13]. This is not always successful. The problem of implicitly defined relations is also addressed in the contouring of functions [12].

The method of interval constraint plotting has the property that the plot shows a *hull* of the function or relation. This property holds independently of rounding errors. In cases where the precision of the underlying floating-point system is insufficient, this limitation shows by the hull being wider than necessary, but it remains a hull. Otherwise, the plotted hull is the smallest one allowed by the resolution resulting from the selected size of pixel. The Java applet *IASolver* [9] designed and implemented by one of us (TH) is based on the method of interval constraint plotting.

In this paper we start by reviewing conventional plotting methods. We contrast it against what we call *raster-scan plotting*, which is extremely simple to implement and can handle any relation. But raster-scan plotting is inefficient, suffers from deformations and is susceptible to rounding errors. To remedy these problems we first introduce the general ideas of interval arithmetic and interval constraints and then show how these can be used to eliminate the problems of raster-scan plotting. The result is what we call *interval constraint plotting*.

2. Conventional Plotting

Conventional techniques for plotting a function $y = f(x)$ usually apply something like the following algorithm:

```
for (x = a; x <= b; x = x + dx)
  plot(x, f(x));
```

In its naive form, `dx` is a constant which means function $y = f(x)$ will be sampled at a predetermined number of equally spaced points. This is not satisfactory for curves that oscillate much more strongly in some areas than in others. In adaptive plotting, `dx` is a variable determined by the program to add more points between the original ones to more accurately draw strongly oscillating curves.

In [6], Fateman identifies undersampling as the major drawback in these conventional methods. Even adaptive sampling does not protect against an injudicious choice of points. The plotter may still choose insufficiently many points to faithfully render part of a curve. This problem is illustrated in Figure 1, produced by *Mathematica 3.0*. The left plot was produced with naive plotting, while the right plot employed adaptive plotting. Notice that the graph becomes erratic when $x \to 0$, even when adaptive plotting is used, as the sampling density cannot adapt to the increasingly rapid oscillation of $\sin(1/x)$. Figure 2 shows the hull of the graph computed by *IASolver* at two resolutions, corresponding approximately to the effective resolutions used in the plots of Figure 1. The area of rapid oscillation near $x = 0$ is represented in these latter graphs as a rectangular block of size $[0, x_0] \times [-1, 1]$ which

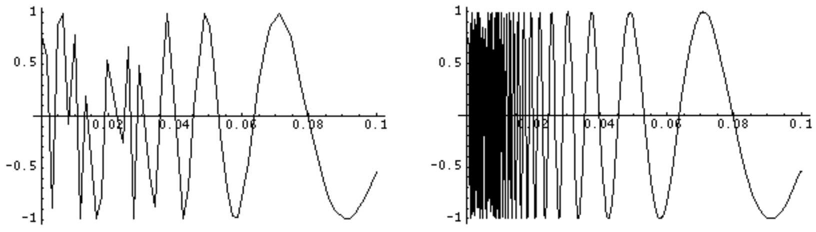

Figure 1. Two plots of $\sin(1/x)$ over the range $[0, 1]$ using *Mathematica 3.0* with non-adaptive plotting on the left and adaptive plotting on the right.

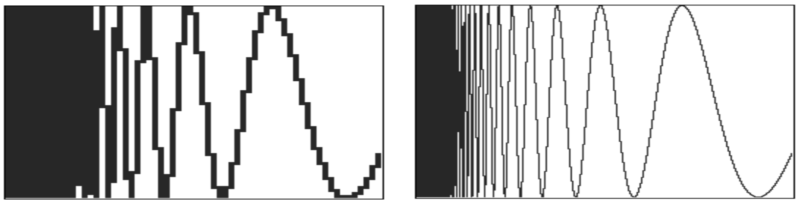

Figure 2. Interval constraint plots of $\sin(1/x)$ using *IASolver* on the range [0,0.1] with a 64x64 grid on the left and a 256x256 grid on the right.

indicates that the function takes values in $[-1, 1]$ when $x \in [0, x_0]$, but provides no other information.

Implicitly defined mathematical relations introduce a more severe problem for conventional plotting techniques. Consider the circle defined by $x^2 + y^2 = 1$. It appears to be necessary to transform this innocent equation to $y = \pm\sqrt{1 - x^2}$ or to $y = \sin\theta \ \wedge \ x = \cos\theta$. The following is a *Matlab* session to plot this circle:

```
>> x = -1:0.001:1; plot(x,sqrt(1-x.^2));
>> hold on; plot(x,-sqrt(1-x.^2));
```

However, such transformations are not always convenient. Consider the implicitly defined relation $\sin(x \cos y) = \cos(y \sin x)$. Isolation of either y or x with respect to the other cannot easily be found. Most plotting programs do have some kind of a contour plotting facility which can be set to plot the constraint $f(x, y) = 0$ by plotting the single $z = 0$ contour for the function $f(x, y) = z$. For example, the following *Matlab* commands generate the contour plot shown in the left half of Figure 3. *Mathematica* and *Maple* offer similar features. The quality of this plot will be discussed later.

```
>> x= 0:0.2:6.2; y= 0:0.2:6.2; [X,Y] = meshgrid(x,y);
>> Z = sin(X.*cos(Y))-cos(Y.*sin(X));
>> contour(x,y,Z,1)
```

The plot on the right half of Figure 3 will be discussed in the next section.

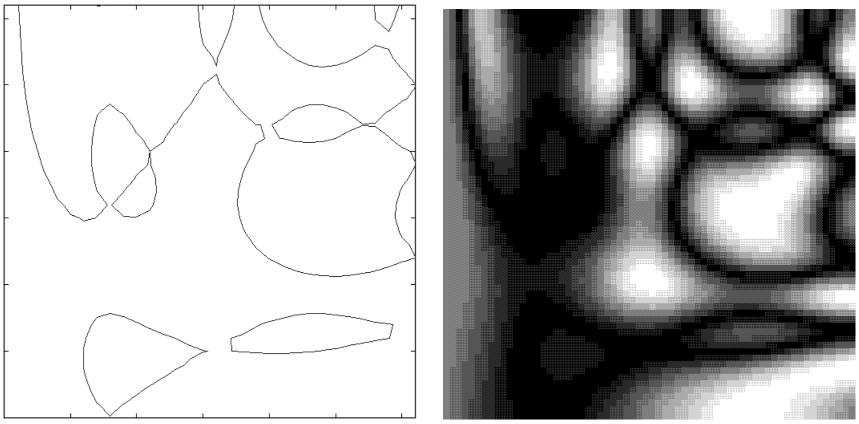

Figure 3. A Contour plot on the left and a Density plot on the right for the function $\sin(x \cos y) - \cos(y \sin x)$ over the range $[0, 2\pi]$. The contour plot was made using *Matlab 5.1*, the density plot was generated by applying the *Mathematica 3.0* function `DensityPlot` to `Abs[Sin[x Cos[y]]-Cos[y Sin[x]]]`

3. Towards Truth in Plotting

3.1. *The Raster-Scan Plotting Algorithm*

One can plot any equality relation $f(x, y) = 0$ by evaluating the relation at each point of a raster. We call this *raster-scan plotting.* It is basically the following:

```
for (x = a; x <= b; x = x + dx)
  for (y = c; y <= d; y = y + dy)
    if (abs(f(x,y)) <= eps) plot(x,y);
```

Inequality relations are handled in the same way. In fact, the condition in the `if` statement can be any boolean expression, a freedom that has been exploited to obtain plots of fractals, where the condition depends on the behaviour of an iteration. Several plotting packages offer a plotting feature in which different shades of gray are used to represent the raster plots for different values of epsilon. For example, the *Mathematica* `DensityPlot` function generates the graph on the right side of Figure 3 for the function $|\sin(x\cos(y)) - \cos(y\sin(x))|$. Though there are several problems with raster-scan plotting, it has the important advantage of being able to handle definitions in functional as well as in relational form. The problems are:

- If the graph is a line, then the same `eps` will give a thicker line in some places than in others. It takes some fiddling to get an acceptable plot. With conventional plotting it is bad enough that the choice of `dx` has such far-reaching consequences. More arbitrariness is introduced with the additional choice of `eps`.

- It is slow: quadratic in the number of points along a linear dimension.
- It does not adapt to the nature of the plot: featureless areas take as much time as those with detail.

By combining the idea of raster-scan plot with interval constraints, we avoid these three problems. The idea is to replace the above code by:

```
for (x = a; x <= b; x = x + dx)
  for (y = c; y <= d; y = y + dy) {
    solve the constraint system:
    f(u,v) = 0 && x <= u <= x+dx && y <= v <= y+dy;
    Paint the rectangle with sides [x,x+dx] and [y,y+dy]
    black or white according to success or failure.
  }
```

This has the following advantages. First, the arbitrary choice of `eps` is no longer needed. Second, a rectangle is plotted white only if the constraint solver can prove that it contains no solutions to the constraint. As a result, *what is plotted in black is a hull of the curve.*

In this way no features are missed in the sense that the hull is shown. Of course, features are missed in the sense that no detail is shown within the elementary rectangles $[x, x + dx] \times [y, y + dy]$. In the following sections we first give a brief introduction to interval constraints and then discuss how *IASolver* optimizes the double for-loop shown here to obtain what appears to be a linear algorithm.

3.2. Brief Introduction to Interval Constraints

In this section, we give an informal introduction to interval constraints and illustrate it with an example.

Let us consider as example the problem of computing x and y in a circle defined as $x^2 + y^2 = 1$. As is clear by inspection, all pairs of feasible x and y values occur in the range $[-1, 1]$. Suppose that we have obtained additional information of x and y lying within $[0.7, 0.8]$. Then an interval constraint system of interest is:

$$x^2 + y^2 = 1 \wedge 0.7 \leq x \leq 0.8 \wedge 0.7 \leq y \leq 0.8 \tag{1}$$

This is a conjunction of three logical formulas related by sharing x and y which denote unknown reals.

The logical formulas are regarded as constraints on the possible values of the unknowns. An interval constraint implementation computes intervals containing all values of which it cannot demonstrate that they are inconsistent. That is, all solutions, if any, are contained within the computed intervals.

An implementation of interval constraints translates the given constraints to primitive constraints. The most commonly used ones are shown in Table 1.

Table 1. Primitive Constraints.

$x + y = z$
$x * y = z$
$x^n = y$ for integer $n \geq 2$
$\exp(x) = y$
$\sin(x) = y$
$\cos(x) = y$
$\tan(x) = y$
$x = y$
$x < y$
$x \leq y$

Complex constraints such as occur in the left-hand side of the first formula of interval constraint system (1) cannot be processed directly by known methods. Instead, the interval constraint system given in equation (1) is translated to primitive constraints. The translation process introduces auxiliary variables x_1, y_1:

$$\begin{aligned} x^2 = x_1 \wedge x_1 + y_1 = 1 \wedge y^2 = y_1 \wedge \\ 0.7 \leq x \wedge x \leq 0.8 \wedge 0.7 \leq y \wedge y \leq 0.8 \end{aligned}$$

We have chosen this interval constraint system because it could occur when applying interval constraint plotting to a circle. At this stage the rectangle $[0.7, 0.8] \times [0.7, 0.8]$ is being considered.

As each of the variables is regarded as an unknown real, it is associated with an interval containing all values of this real that occur in any solution. To solve such a system according to interval constraints, one starts with intervals large enough to contain all solutions of interest. Then one iterates among the primitive constraints, reducing interval to subinterval as far as necessary to remove values that are inconsistent with the constraint under consideration.

Consider, for example, constraints of the form $u + v = w$. Suppose that the intervals for u, v and w are $[0, 2], [0, 2]$ and $[3, 5]$ respectively, then all three intervals contain inconsistent values. Now $v \leq 2$ and $w \geq 3$ imply that $u = w - v \geq 1$. Hence the values less than 1 for u are *inconsistent*, in the conventional sense of the word: it is inconsistent to believe the negation of the conclusion of a logical implication if one accepts the premises. Similar considerations rule out values less than 1 for v and values greater than 4 for w. Removing all inconsistent values from the *a priori* given intervals leaves the intervals $[1, 2]$ for u and v and $[3, 4]$ for w. The new bounds 1 and 3 are computed according to the rules of interval arithmetic and require rounding direction to be set appropriately. Thus interval constraints depends on interval arithmetic [1, 8, 10].

An interval constraint system performs such a constraint contraction for each of the primitive constraints. Because typically constraints share variables, constraint contraction usually has to be performed multiple times on any given constraint: every time another constraint causes the interval for a variable to contract, all constraints containing that variable have to be contracted again. Because changes are always contractions and because interval bounds are floating-point numbers, a finite number of contractions suffices to reach a state where all constraints yield a null contraction. The constraint relaxation algorithm terminates when this is found to be the case.

For example, let us solve the system (1) with the *IASolver* implementation, starting with two-way infinite intervals for all variables. The system stabilizes with an interval contained in $[-1, 1]$ for x, and similarly for y. Adding the additional constraints that $x \in [0.7, 0.8]$ and $y \in [0.7, 0.8]$ results x and y to be narrowed to $[0.7, 0.7141428429]$.

Thus in interval constraint plotting, the original rectangle $[0.7, 0.8] \times [0.7, 0.8]$ undergoes a considerable reduction to a subset of $[0.7, 0.72] \times [0.7, 0.72]$.

As the consistency method only removes inconsistent values and may not remove all such values, it may be that the resulting intervals contain no solution. Thus, results in the consistency method have the meaning: *if* a solution exists, then it is in the intervals found.

Some references to interval constraints are [4, 5, 15, 16].

3.3. Interval Constraint Plotting

Conventional plotting, because of the single `for` loop, is linear, but restricted to functions. Raster-scan plotting handles relations, but is quadratic. It might be thought that quadraticity is inherent in the ability to handle relations. This is not so. Interval constraint plotting is able to dispatch large areas of the plotting space due to failure of a single interval constraint system. Thus, interval constraint plotting is adaptive in a similar sense to adaptive plotting of functions: featureless areas of the plotting space require less computing effort.

An algorithm that exploits this adaptivity might look as follows:

```
void function plot(Rectangle rect) {
  create and solve the interval constraint system
  "r(x,y) && point (x,y) is in rect";
  if (failure) {paint rect white; return;}
  if (rect small enough) {paint rect black; return;}
  split rect into four rectangles: nw, sw, se, ne;
  plot(nw); plot(sw); plot(se); plot(ne);
}
```

Execution of this algorithm follows a quad tree [17], but only as far as necessary: interval constraint systems corresponding to large areas can fail and can then be painted in one step. How significant a savings does this represent?

To get at least a partial answer to this question, we instrumented the interval constraint plotting algorithm with a counter to record how many nodes of the complete quadtree were visited. As relation we chose $\sin(x * \cos(y)) = \cos(y * \sin(x))$. As initial rectangle we chose one of size 2π by 2π. We ran the program with 10 recursive levels of subdivision, corresponding to grids of size 1x1, 2x2, 4x4, ..., 1024x1024. The upper line in Figure 4 shows the total number of tree

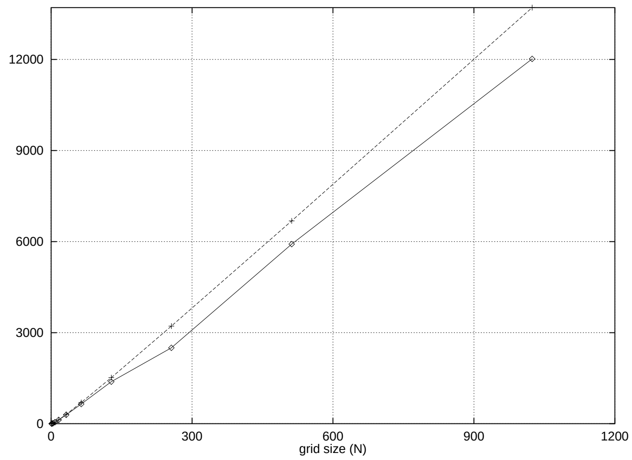

Figure 4. Number of nodes visited as function of number (N) of subdivisions. Upper line is for the constraint $\sin(x\cos(y)) = \cos(y\sin(x))$, lower line is for the constraint $x^2 + y^2 = 1$. These two lines are almost linear with slopes of about 14 and 12, respectively.

nodes visited for each grid size from 1 to 1024. The lower line in that Figure shows the corresponding data for the relation $x^2 + y^2 = 1$ with x, y initially in the interval $[-1, 1]$. These data suggests that the growth rate is linear; a worthwhile improvement over the quadratic nature of the version directly derived from the raster-scan method.

There is also a heuristic argument for linearity in the interval constraint plotting method. First observe that the graph of a monotone function that passes through an $N \times N$ rectangular grid of cells can intersect at most $3N$ of the cells. (Indeed, such a graph will cross at most N horizontal grid lines and at most N vertical grid lines before leaving the grid. Thus it can enter into the interior of at most $2N$ grid cells. If A of these crossings are grid cell vertices, i.e., intersections of horizontal and vertical grid lines, then each such crossing accounts for an intersection with at most three new grid cells. Thus, the graph can intersect at most $3A + 2(N - A) \leq 3N$ grid cells.) Thus, if we assume that the constraint solver will fail on all cells that do not intersect the graph, and if we assume that the relation being plotted eventually becomes locally monotone after a certain number of subdivisions, then all further refinements of the graph will require linear execution time in the number of new subdivisions.

4. IASolver

IASolver is a Java applet that provides the dual capabilities of solving non-linear constraints and interactively controlled interval constraint plotting. The system has a graphical user interface allowing the user to enter constraint expressions, to initiate interval stabilization, to view the resulting intervals, and to plot according

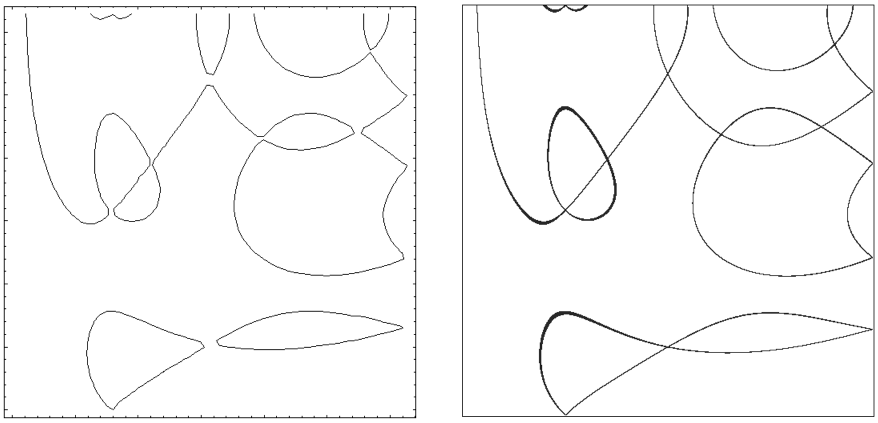

Figure 5. Constraint plots of $\sin(x\cos y) = \cos(y\sin x)$ over the range $[0, 2\pi]$ in *Mathematica 3.0.* (on the left) and *IAsolver* (on the right)

to selected variables. A large number of constraints can be handled, in a large number of variables. Any two of these can be selected as X and Y coordinates for plotting.

For plotting, resolution can be selected up to a maximum of 1024 subdivisions of the plotting window in each linear dimension. Figure 2 shows two resolutions for the constraint $y = \sin(1/x)$. Any subwindow can be selected using the mouse and made into the plotting window, giving the effect of zooming. In this way one can proceed to a resolution where successive floating-point numbers become distinguishable.

To make the interval constraint plotting easier to follow by the user, a number of modifications to the recursive quadtree algorithm of the previous section have been made. Instead of proceeding depth-first, *IASolver* proceeds level by level. In addition, it keeps the non-failed rectangles in a list. As a result, with nonpathological curves, one sees the contractions proceed along the curve.

Let us now review the performance of *IASolver* on the examples introduced in section 2. We have already seen in our previous example of $\sin 1/x$ (Figure 2) how *IASolver* can incrementally reveal details of a curve Compared with the graph in Figure 1 plotted by *Mathematica 3.0*, *IASolver* not only portrays appropriately the problematic region near $x = 0$, but also allows more interesting features of the curve to be displayed at a higher resolution.

Figure 5 shows a comparison of a *Mathematica* contour plot (on the left) with an *IAsolver* interval constraint plot (on the right) of the relation $\sin(x\cos y) = \cos(y\sin x)$ over the range $[0, 2\pi]$. There exist a certain number of discrepancies where some regions are disjoint in the *Mathematica* plot, but not in the *IAsolover* plot. By analyzing the solution set locally at the various intersection points we can determine that the graph produced by IAsolver not only contains the solution set, but is topologically correct as well. The *Mathematica* graph has neither of these properties.

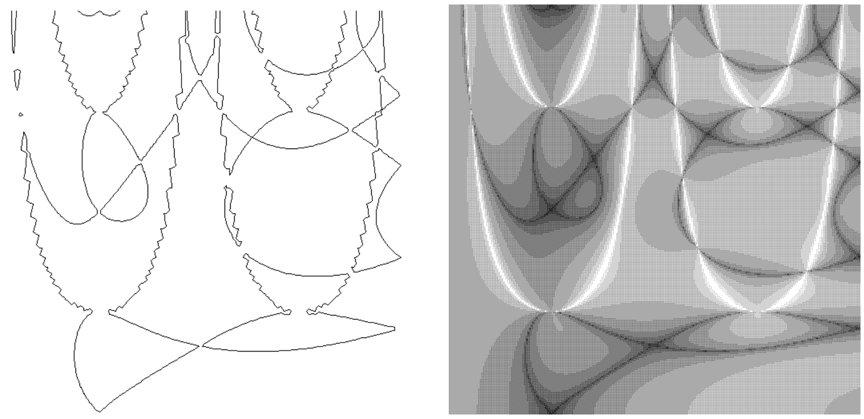

Figure 6. Plots of $\sin(x \cos y)/\cos(y \sin x) = 1$ over the range $[0, 2\pi]$ in *Mathematica 3.0.* using contour plotting on the left and density plotting on the right. The *IAsolver* plot of this relation is identical to the one in Figure 5

Constraints involving singular functions can create problems for contour methods. For example, in Figure 6 we see the result of plotting the solutions to $1 = \sin(x\cos(y))/\cos(y\sin(x))$ with *Mathematica*'s ContourPlot on the left and DensityPlot on the right. Observe that the contour plot incorrectly contains a rough approximation of the graph of singularities of the function. The DensityPlot graph is somewhat better since it distinguishes between the singular locus in white and the solution set in black (note that some tuning in the choice of gray scale map was required to get this particular plot), but it still provides only an approximation (with no specified error bound) of the solution set. The graph produced by IAsolver for this constraint is identical to the graph in Figure 5 because in interval constraints multiplications and divisions are symmetrical components of the same ternary product constraint.

5. Related Work

Suffern [12] was concerned with the contouring of functions of two variables. This is an instance of an implicitly defined relation. Suffern uses quadtrees to avoid the quadratic behaviour in the raster-scan plotting method. The illustrations suggest $\mathcal{O}(n)$ behaviour, but the paper is not concerned with this aspect. The method uses discrete points in a grid, hence is subject to the problem of undersampling. There is no guarantee of obtaining a hull of the contour.

Several papers [6, 2, 11] have used interval methods in plotting. Fateman [6] implemented interval arithmetic for the plotting of functions in *Mathematica* to enhance its plotting facilities. He pointed out that interval methods can give the hull property for graphs of functions, a property he referred to as "honest" plotting.

6. Conclusions

Interval constraints represents a new approach to numerical computation. It has been shown to be especially effective for solving large numbers of nonlinear equations and for nonconvex global optimization. Its basic operation, constraint contraction, relies on interval arithmetic. Its implementations have benefited from the experience gathered by interval arithmetic, especially as embodied in the IEEE standard for floating-point arithmetic, which provides the infinities and directed rounding.

Because interval constraints only eliminates inconsistent values for the variables concerned, it can never miss solutions (assuming, of course, that the underlying interval constraint solving hardware/software system is correctly implemented). Hence, when the outcome includes an empty interval for a variable, there is no doubt that there is indeed no solution. In other words, there are no "false negatives" in interval constraints. On the other hand, stabilization with all intervals nonempty only means that if there are solutions, they must be in the remaining intervals. In other words, "false positives" are possible.

In our experience, the latter only arises in situations where rectangles are too large or where the precision of the underlying floating-point system is inadequate. The famous horrors of numerical analysis, the Wilkinson polynomial and other examples in [7] tend to show up as false positives in interval constraints.

In interval constraint plotting the asymmetry between absence of false negatives and the possibility of false positives translates to the fact that a hull of the true plot is shown. False positives become less likely the finer the subdivision of the plotting area. Because of this it is encouraging that computation time appears to grow linearly with the number of subdivisions of the plotting axes.

The asymmetry between absence of false negatives and the possibility of false positives may appear to be a fundamental limitation. In plotting this hardly appears to be the case. Suppose we are interested in the $z = 0$ contour of a continuous function $z = f(x, y)$. In that case we should invoke interval constraint plotting twice: once for $f(x, y) < 0$ and once for $f(x, y) > 0$. In each case the white areas, where interval constraints failed, guarantee the absence of solutions of the equality. Rounding errors have the effect of making these areas slightly too small. Thus, for contours, interval constraint plotting gives with certainty an area for sure negative values and an area for sure positive values. In between there is a narrow area of uncertainty. With the usual proviso, this narrow area is a perfectly precise rendering of the contour.

IASolver does not yet allow such use: it only gives a plot for a single interval constraint system (but, of course, with any number of constraints). It cannot yet superimpose plots arising from different interval constraint systems. Also, it is quite slow, because the Java virtual machine is. Yet it is fast enough to allow enjoyable experimentation, which the reader is hereby invited to do.

Acknowledgments

We gratefully acknowledge the Natural Science and Engineering Research Council of Canada for generously providing research facilities. Many thanks to Ying Luo for helpful conversations.

References

1. Alefeld, G. and J. Herzberger, *Introduction to Interval Computations*, Academic Press, 1983.
2. Avitzur, R., O. Bachmann and N. Kajler, "From Honest to Intelligent Plotting," *Proc. of ISSAC'95*, July 1995, Montreal, Canada, pp. 32–41, ACM Press.
3. Bajaj, C., C. Hoffman, J. Hopcroft and R. Lynch, "Tracing Surface Intersections," *Computer Aided Geometric Design*, 5, 1988, pp. 285–307.
4. Benhamou, F. and W. Older, "Applying Interval Arithmetic to Real, Integer, and Boolean Constraints," *Journal of Logic Programming*, 32(1), 1997, pp. 1–24.
5. BNR, *BNR Prolog User Guide and Reference Manual*, Software Engineering Center, Bell-Northern Research, Ottawa, Canada, 1988.
6. Fateman, R., "Honest Plotting, Global Extrema and Interval Arithmetic," *Proc. of ISSAC'92*, July 1992, Berkeley, CA, pp. 216–223, ACM Press.
7. Forsythe, G., "Pitfalls in Computation, or Why a Math Book isn't Enough," *Amer. Math. Monthly*, 77, 1970, pp. 931-956.
8. Hansen, E., *Global Optimization Using Interval Analysis*, Marcel Dekker, 1992.
9. Hickey, T., *IASolver*, Java applet accessible via `http://www.cs.brandeis.edu/~tim`.
10. Moore, R., *Interval Analysis*, Prentice-Hall, 1966.
11. Snyder, J., "Interval Analysis for Computer Graphics," *Computer Graphics*, July 1992, pp. 121–129.
12. Suffern, K., "Quadtree Algorithms for Contouring Functions of Two Variables," *The Computer Journal*, 33, 1990, pp. 402–407.
13. Timmer, H., "Analytic Background for Computation of Surface Intersections," Douglas Aircraft Company Technical Memorandum CI-250-CAT-77-036, April 1977.
14. Tupper, J., *Graphing Equations with Generalized Interval Arithmetic.* M. Sc thesis, Department of Computer Science, University of Toronto, 1996.
15. van Emden, M., "Value Constraints in the CLP Scheme," *Constraints: An International Journal*, 2, 1997, pp. 163–183.
16. Van Hentenryck, P., L. Michel and Y. Deville, *Numerica: A Modelling Language for Global Optimization*, MIT Press, 1997.
17. Williams, R., "The Goblin Quadtree," *The Computer Journal*, 31, 1988, pp. 358–363.