Numer Algor (2009) 52:605–628
DOI 10.1007/s11075-009-9303-2

ORIGINAL PAPER

An adaptive algorithm for efficient computation of level curves of surfaces

Dimitri Breda · Stefano Maset · Rossana Vermiglio

Received: 7 January 2008 / Accepted: 22 May 2009 /
Published online: 1 July 2009
© Springer Science + Business Media, LLC 2009

Abstract A new efficient algorithm for the computation of $z =$ constant level curves of surfaces $z = f(x, y)$ is proposed and tested on several examples. The set of z-level curves in a given rectangle of the (x, y)-plane is obtained by evaluating f on a first coarse square grid which is then adaptively refined by triangulation to eventually match a desired tolerance. Adaptivity leads to a considerable reduction in terms of evaluations of f with respect to uniform grid computation as in Matlab®'s `contour`. Furthermore, especially when the evaluation of f is computationally expensive, this reduction notably decreases the computational time. A comparison of performances is shown for two real-life applications such as the determination of stability charts and of $\varepsilon-$pseudospectra for linear time delay systems. The corresponding Matlab code is also discussed.

Keywords Level curves · Adaptive computation · Contour plot

1 Introduction

In this paper we face the problem of computing a set of $z =$ constant level curves of a surface $z = f(x, y)$ where, possibly, f does not have an explicit

D. Breda (✉) · R. Vermiglio
Dipartimento di Matematica e Informatica, Università degli Studi di Udine,
via delle Scienze 208, 33100 Udine, Italy
e-mail: dimitri.breda@dimi.uniud.it

R. Vermiglio
e-mail: rossana.vermiglio@dimi.uniud.it

S. Maset
Dipartimento di Matematica e Informatica, Università degli Studi di Trieste,
p.le Europa 1, 34127 Trieste, Italy
e-mail: maset@univ.trieste.it

form but rather it can be numerically evaluated for any choice of x and y in a given rectangular region of the (x, y)-plane.

Although the problem could appear simple at a first sight, it hides nontrivial arguments and peculiarities which have to be considered and exploited if the final target is to obtain a certain accuracy in the level curves with the least possible computational effort. Think for instance at two-parameters robust analysis which very often raises in control theory and automation: here the interest is in the asymptotic stability analysis of control(led) systems with uncertain parameters. The complete stability map in the parameters plane is the set of level curves $f(x, y) = z$ where f is a "stability indicator", e.g. a function giving the (real part of the) rightmost eigenvalue governing the system dynamics, in which case $z = 0$. Hence typically f corresponds to an exact (or numerically approximated) eigenvalue problem, possibly of large dimension (e.g. space-discretized partial differential equations), and its computation at one point (x, y) could be substantial.

It is clear from the previous example that the simple but natural idea of computing f in as many regularly spaced point (x, y) as required in order to get accurate level curves could reveal itself an enormous waste of computational resources such as CPU time and memory storage. Why should we compute f almost everywhere if our interest runs only along a finite set of curves? Of course these curves represent the unknowns of our problem, but this motivated us to search for new algorithms which can adaptively know how to get closer to these curves forgetting about the regions far away from them.

A new algorithm following the above lines is proposed and described in this work. In particular it is based on an adaptive triangulation scheme instead of using uniformly spaced grids of points as done in Matlab's `contour`. Moreover it uses secant method instead of linear interpolation to eventually detect points on a level curve once this has been located inside a portion of the (x, y)-plane. Finally, other tricks or heuristic choices are included in order to overcome difficult situations which in general arise when a surface intersects a plane.

The new algorithm is implemented in a Matlab package (freely available from Netlib, http://www.netlib.org/numeralgo/na27) which is tested on several examples and compared to Matlab's `contour` on two real-life applications which are known to be computationally expensive, i.e. the determination of the stability chart and of the $\varepsilon-$pseudospectra of linear systems of Delay Differential Equations (DDEs).

2 Test cases

In order to describe the features of both Matlab's `contour` and our algorithm, in the sequel `level`, we use the following constructed surface functions:

$$z = f(x, y) = \frac{\sin(\pi x) - 1}{2} + \frac{x + 1}{\sqrt{2}} - y \tag{1}$$

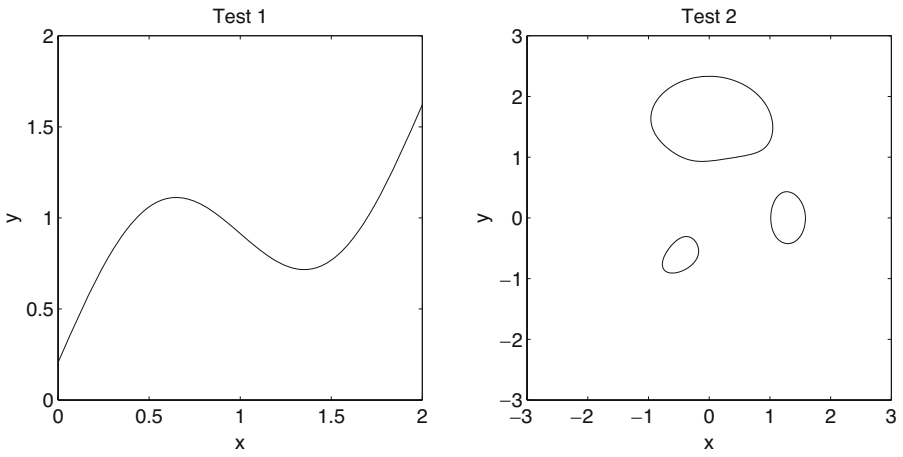

Fig. 1 $z = 0$ level curve for Test 1 (*left*) and $z = 3$ level curves for Test 2 (*right*)

with $(x, y) \in [0, 2] \times [0, 2]$, and

$$z = f(x, y) = 3(1 - x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right) e^{-x^2-y^2} - \frac{1}{3} e^{-(x+1)^2-y^2} \tag{2}$$

with $(x, y) \in [-3, 3] \times [-3, 3]$. The first one (indicated as "Test 1" in the sequel and in the relevant software) is an ad hoc case built in order to get an explicit form of the exact $z = 0$ level curve, which is simply given by

$$y = \frac{\sin(\pi x) - 1}{2} + \frac{x + 1}{\sqrt{2}}, \quad x \in [0, 2],$$

and it is shown in Fig. 1 (left). The second one (indicated as "Test 2" in the sequel and in the relevant software) is the function `peaks` used in Matlab as an example for surface and contour plots. The set of $z = 3$ level curves is shown in Fig. 1 (right).

No matter which level $z =$ constant we are interested in, in the rest of the paper we always refer to the case $z = 0$ by translation along the vertical axis, i.e. by considering $f(x, y) -$ constant instead of $f(x, y)$. In this sense, the discriminant to say that a point of the surface is above or below the desired level is the sign of its f value, positive or negative, respectively.

3 Matlab's contour

In this section we give a brief description of the strategy behind the `contour` function used in Matlab. For further details we refer the reader to the relevant online documentation.

The `contour`'s method is based on the following steps. A uniform grid of equally spaced points is set on the given rectangular region of the (x, y)-plane.

This grid divides the region into rectangular cells and f is evaluated at the four vertices of each cell. Then, if the sign of f at the four vertices changes it means that a segment of level curve is crossing the cell at two edges. The two crossing (or zero) points are determined by linear interpolation of the corresponding vertex values. Finally, the segment of level curve is approximated by joining the zero points with a straight line. To the best of the authors' knowledge there is no further rule behind this Matlab code.

With such an algorithm, it is obvious that the final accuracy on the level curve depends on the grid size and it seems there is no other reasonable way to measure the accuracy of a curve lying on a plane but inscribing it into a bounded portion of the plane itself. In this sense, we can say that each point on the resulting level curve is correct within a given tolerance TOL if, for instance, the (absolute) length of the longest edge of the rectangles forming the uniform grid is less than TOL. The (relative) tolerance ensured by the default 60×60 `contour`'s grid can be changed in order to increase this final accuracy.

Matlab's `contour` imposes the evaluation of f on a set of uniformly distributed points independently on the actual location of the level curves. Hence it inevitably calculate many f values which are useless to determine the curves, and remember that f is possibly expensive to evaluate. To reduce the amount of these "useless" points, it seems better to begin with "few" points and then perform some sort of adaptive refinement as explained in the next section.

4 Adaptive triangulation

A first natural adaptive refinement is obtained by starting with a coarse rectangular grid, and refine each cell with sign changes at the vertices by dividing it into four further rectangular cells. The process can be iterated until the final required accuracy is matched, i.e. the diagonal of each rectangle is less or equal than a given tolerance TOL. In this way, the closer we are to a level curve, the higher is the number of f evaluations. We call this *adaptive rectangular refinement*, and each cell subdivision requires five new computations of f, i.e. all the edges mid points and the cell center, see Fig. 2 (left).

In `level` we introduce a further and substantial improvement called *adaptive triangular refinement*. Each rectangular cell with sign changes of the starting coarse grid is refined by dividing it into four triangles by using only one new evaluation of f, i.e. the center of the cell, see Fig. 2 (right). The refinement of a new triangle is obtained by dividing it into two further triangles by the height relevant to the longest edge, at cost of one new f evaluation.

For regularity reasons which will be clear in Section 6, and without loss of generality, in the sequel we consider a starting grid of squares (and not rectangles). Therefore, each square subdivision results in four isosceles rectangular triangles and each of these is divided again into two isosceles rectangular triangles by the height relevant to the hypotenuse.

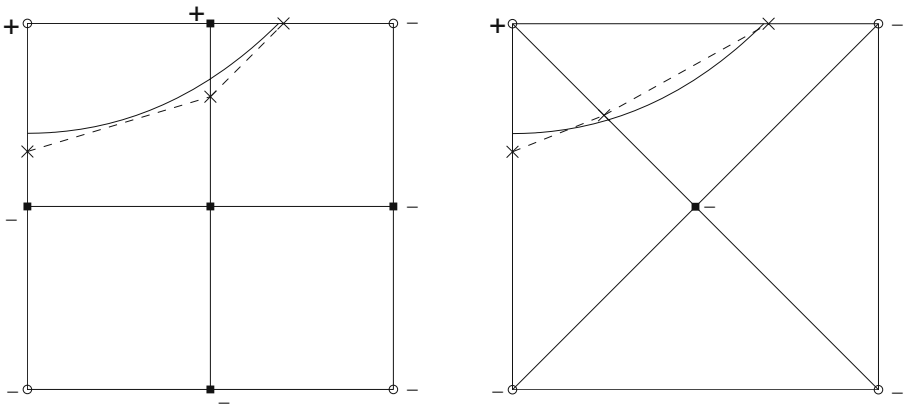

Fig. 2 New f evaluations (*black squares*) with respect to old ones (*empty circles*) with adaptive rectangular (*left*) and triangular (*right*) refinement and example of exact (*solid line*) and approximated (*dashed line*) level curve

Similarly to what described in Section 3, in the case of triangulation we choose to control the (absolute) length of the cathetus of each triangle, i.e. a triangle with sign changes is refined until the length of its cathetus is less than a prescribed tolerance TOL, in the sequel intended as "final accuracy". This latter is also the main tolerance value given in input.

Although the triangulation turns out to be a more complex structure than what can be obtained by square refinement, the computational advantage is easily shown with the following argument. Suppose we start from a square cell of area A and we need to reach a single cell of area $a < A$. Then, the minimum number n_f of new evaluations of f necessary to reach a from A is given by

$$n_f = 5 \left\lceil \frac{\log\left(\frac{A}{a}\right)}{\log 4} \right\rceil$$

using squares and by

$$n_f = 2 \left\lceil \frac{\log\left(\frac{A}{a}\right)}{\log 4} \right\rceil - 1$$

using triangles, where $\lceil x \rceil$ denotes the minimum integer p such that $p \geq x$. Therefore, the averaged computational gain is more than half using triangulation.

The adaptive triangulation strategy produces a set of triangles in the (x, y)-plane that certainly contains segments of the (still unknown) level curves. In the next section we explain how to approximate (and hence eventually plot) these segments.

5 Level curve location

The problem is the following: given a (triangular) cell with sign changes along two edges, locate the segment of level curve crossing the cell or, in other terms, find the zero of the function f along each edge with sign change. As in `contour`, linear interpolation can be used obtaining a first approximation of the exact zero point. One could improve by applying the more efficient and well-established Newton's method for zeros of nonlinear functions [6], but this requires the knowledge of the first derivative of f. The latter is not always known, especially if f is not given in explicit form, but only computable for every choice of x and y, as in our case. An approximation to f' could be found, but this would mean extra computational cost, which is opposite to our goal. Hence we decide to apply the secant method [6] which needs two initial guesses (already known having the two vertex f values) but not f'. Moreover, `level` implements a slight variation known as *regula falsi* [6] that ensures convergence inside the edge with no extra cost. In practice, this is like applying successive steps of linear interpolation each between the f value corresponding to the new point and the previous one with opposite sign.

Once the two zero points are found, they are joint with a straight line. In Fig. 3 we show the situation for a cell: the surface is represented with solid lines, the linear interpolation by dashed lines and the triangular cell by dotted lines. The left picture shows a "nice" situation in which linear interpolation could give good results, but the right one shows a worse, but more general, case.

In general we get sufficient accuracy with three or four secant iterations, i.e. one or two new f evaluations for each edge. This extra cost is largely

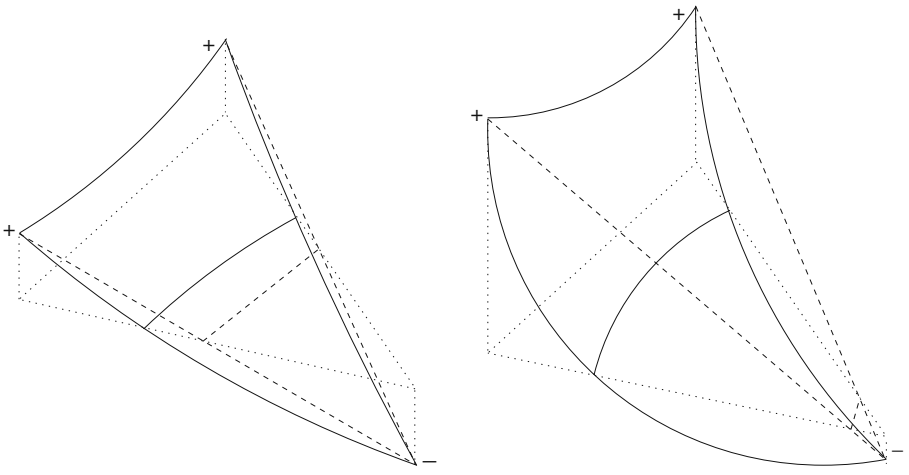

Fig. 3 Location of a level curve segment on a triangular cell by linear interpolation

compensated by the strong reduction obtained by triangulation with respect to uniform square grid.

To resume, we propose first to substitute the uniform square grid with an adaptive triangulation in order to detect a set of small cells where the level curve lies and second to change the linear interpolation with the secant method to locate accurately the segments of level curve inside these cells. A detailed analysis should suggest how to choose the density of the starting coarse square grid and how deep to go with the triangulation to eventually start with the location via the secant method. This depends much on the nature of f. In `level` we choose a fixed compromise (see Section 8 for the details) among these three phases which revealed good on average. So the size of the initial squares, of the smallest triangles and the tolerance (and maximum number) of the secant iterations are fixed a priori, although in the implemented code the user is allowed to modify the default values through optional inputs.

Nevertheless, other important questions are to be faced. Some of them are congenital of the adaptive triangulation, some are not. For instance we refer to the fact that multiple evaluations between neighboring cells easily occur when using adaptive strategies, or to the fact that a segment of level curve can cross (or lies inside) a cell without generating a sign change at its vertex f values. We address these and other problems in the following sections.

6 Multiple evaluations and information storage

Consider two neighboring square cells and suppose that one has already been refined, i.e. f has been evaluated at least at the mid point of each edge (plus the cell center). When we proceed to refine the second one, the value of f at the mid point of the common edge is already known from the refinement of the previous cell. Hence we should absolutely avoid its computation again. Since this can occur quite often, if we do not pay attention the final result could be a almost doubled computational cost, i.e. we risk to loose all the gain given by the adaptive strategy.

The solution is quite easy using squares: it is enough to store all the evaluations of f in a (possibly large and sparse) matrix whose entries indices are related to the grid coordinates in the (x, y)-plane. Every time a new evaluation of f is needed, we first check the corresponding entry of this matrix to see if it is empty or not. The dimension of the matrix depends on that of the rectangular region to scan and on the final accuracy required, i.e. the size of the smallest square, and hence it can be determined in advance.

If we now turn our attention to adaptive triangulation, the same idea might appear not so suitable. But on the contrary, a matrix can be associated to each square of the starting coarse grid and since it is a square, the triangulation (by isosceles rectangular triangles) leads to a regular distribution of the grid points and their (x, y) coordinates can be directly associated to the matrix entries indices.

In detail, if l_s is the length of the starting square and l_t is the cathetus of the final smallest triangle, the maximum number of possible subdivision is the minimum integer n such that

$$\frac{l_s}{(\sqrt{2})^n} \leq l_t,$$

i.e.

$$n = \left\lceil 2 \log_2 \left(\frac{l_s}{l_t} \right) \right\rceil.$$

Observe (Fig. 4) that each possible new vertex belongs to a $d \times d$ uniform grid of equi-spaced points with separation l_g where

$$d = 2^m + 1,$$

$$l_g = \frac{l_s}{2^m}$$

and

$$m = \left\lfloor \frac{n+1}{2} \right\rfloor$$

where $\lfloor p \rfloor$ denotes the largest integer q such that $q \leq p$. We use a $d \times d$ matrix S in such a way that if the vertex has coordinates (x, y) and $f(x, y) = z$, the corresponding matrix entry is $s_{ij} = z$ with

$$i = \frac{y_{\max} - y}{l_g} + 1, \quad j = \frac{x - x_{\min}}{l_g} + 1,$$

where $(x_{\min}, y_{\max})$ are the coordinates of the left-top vertex of the square cell. In this way, when a vertex is introduced by a further subdivision of a triangular cell, its f value can be recovered from the matrix S whenever this vertex has already been computed for a neighboring cell. For instance in Fig. 4, the subdivision of the cell T_1 does not require the evaluation of the subdivision

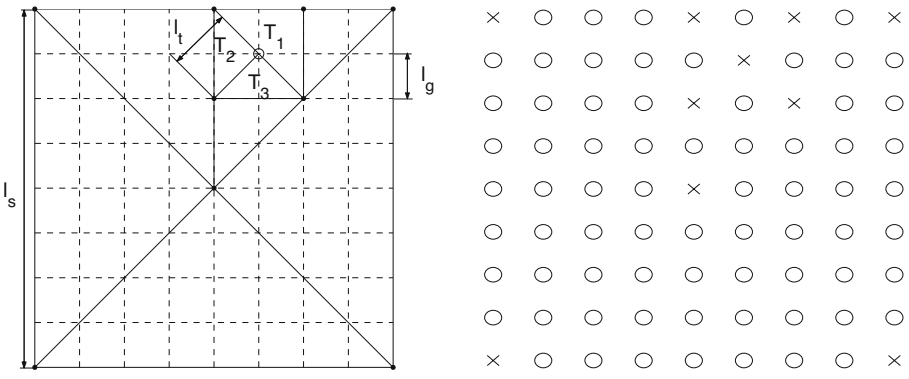

Fig. 4 Example of square cell subdivision (*left*) and its matrix representation (*right*)

vertex (∘) since this is already computed for the cells T_2 and T_3. Moreover, since not all the nodes of the square grid are necessarily vertices of triangular cells, i.e. there is no need to know their f value, the matrix S is usually sparse and therefore its storage is cheap.

6.1 Scanning the starting grid

The algorithm starts by defining a first coarse square grid on the rectangular region $[X_{\min}, X_{\max}] \times [Y_{\min}, Y_{\max}]$ of the (x, y)-plane. Let l_s be the size of the grid square, determined in order to locate at least p squares along the shortest edge, with p a given positive integer (see Section 8 for details). To cover all the region with an integer number of squares we possibly enlarge $X_{\max}$ and $Y_{\max}$ to

$$\overline{X}_{\max} = X_{\min} + n_x l_s$$

and

$$\overline{Y}_{\max} = Y_{\min} + n_y l_s,$$

respectively, where

$$n_x = \left\lceil \frac{X_{\max} - X_{\min}}{l_s} \right\rceil$$

and

$$n_y = \left\lceil \frac{Y_{\max} - Y_{\min}}{l_s} \right\rceil.$$

So n_x and n_y are the number of squares along the horizontal and vertical edges of the region, respectively. The analysis starts from the left-top square cell and goes on towards the right and bottom directions, i.e. the usual reading/writing ones (Fig. 5, left).

Once a square cell has to be refined (the "current" cell, S_c in Fig. 5, right), its four f vertex values are stored into the matrix S which is passed in input to the refinement function as will be clear in Section 8. As for the triangular

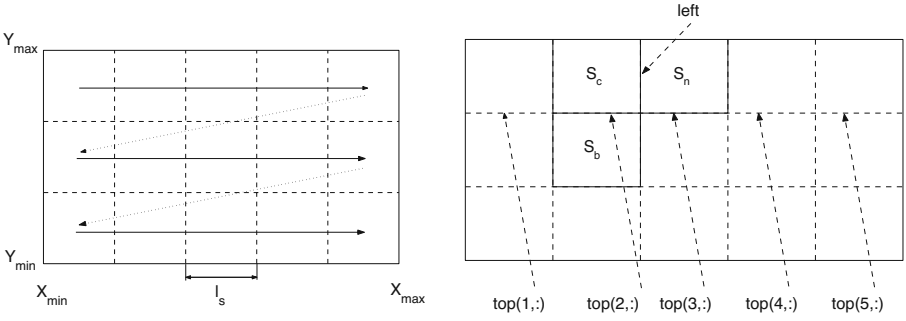

Fig. 5 Example of starting square grid

cells, also the square ones share some vertices. Surely the right, respectively bottom, vertices of the current square cell are the same as the left, respectively top, ones of the "next" square (S_n in Fig. 5, right), respectively "bottom" (S_b in Fig. 5, right). But there might be more vertices in common generated by the triangulation. Hence, to avoid any kind of possible multiple f evaluation, every time a square cell is refined, all the new vertices created along the right, respectively bottom, edge of the refined square cell are stored in a d-vector $left$ and in a $n_x \times d$ matrix top, respectively. The reason of this is the following. Since the square grid is scan towards the bottom row by row, and each row is scan towards the right, the right edge of S_c is passed directly to the left one of S_n which is the next to be refined. Hence a d vector is enough as auxiliary vector to be passed to the next cell. Opposite, the bottom edge of S_c is the top one of S_b which will be possibly refined after n_x steps. Hence the bottom edges of all the square cells of a whole row must be stored for the next row and a $n_x \times d$ matrix is necessary. The i-th, $i = 1, \ldots, n_x$, row of this matrix is filled with the bottom edge of the i-th square cell according to its position along the row of the grid. The refinement function provides to update the vector $left$ and the row of the matrix top which are used next. This applies with some attention when the current cell is the last one of a row or even the right-bottom one.

7 Further refinement

In Section 4 we assumed to refine a (square or triangular) cell when a sign change in the f vertex values occurs along two edges. Indeed, this condition is sufficient to state that a segment of level curve is crossing the cell. But it is not necessary as explained next.

7.1 The slope test

If one looks at Fig. 6, it soon realizes that a segment of level curve may cross a cell, or completely lie inside it, even if there is no sign change at the f vertex values. Also in this case a cell refinement is necessary, but the question is how to detect this possibility.

To this aim consider an edge $\overline{p_1 p_2}$ with values $f_1 = f(p_1)$ and $f_2 = f(p_2)$ of the same sign at the vertices $p_i = (x_i, y_i)$, $i = 1, 2$ (Fig. 7). We check the possibility that a segment crossing exists, i.e. there exists a point $(x, y) = p \in \overline{p_1 p_2}$ with $f(p) = 0$, measuring the minimum slope at which p is reached simultaneously from f_1 and f_2. This slope is given by

$$s = \tan \alpha = \frac{|f_1 + f_2|}{l}$$

where l is the length of the edge. Then we set a tolerance parameter, say $\mathrm{T_p}$, and if $s \geq \mathrm{T_p}$ for all the edges of the cell, then there is no need to refine because the f vertex values are "too far" from zero with respect to the length of the edge. Indeed, this is not a sufficient condition to exclude the refinement, but at

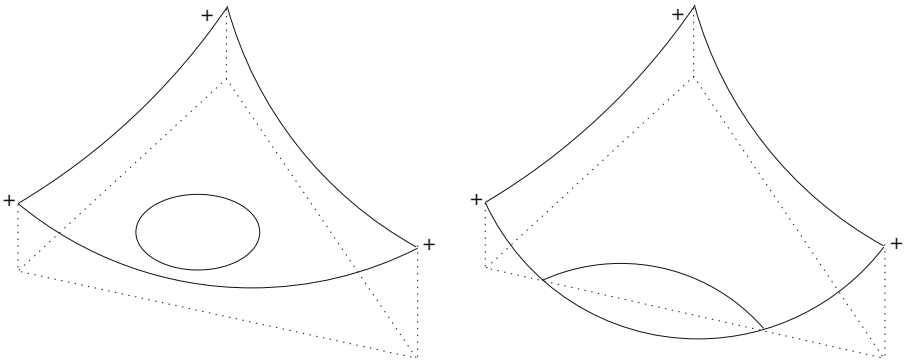

Fig. 6 Examples of level curves not generating sign changes at the vertices of the cell

least it is a good indicator if T_p is chosen correctly. In the implementation we set two values for this tolerance, i.e. $T_p = T_s$ for square cells and $T_p = T_t$ for triangular cells.

A couple of other particular situations are treated in the following sections

7.2 Adaptive curvature determination

So far we are able to determine a set of cells where the level curve lies, and to locate it using a set of segments whose vertices are given via the secant method along the cell edges. As explained in Section 5, few secant iterations allows to detect with good accuracy the zero points along the two cell edges with sign change. But looking at Fig. 8 (left) it is clear that all this accuracy is lost inside the cell if the segment shows a large curvature, and this sounds like unreasonable.

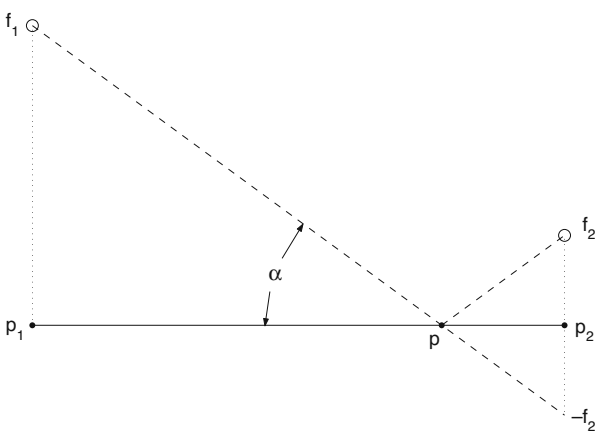

Fig. 7 Slope test on a cell edge

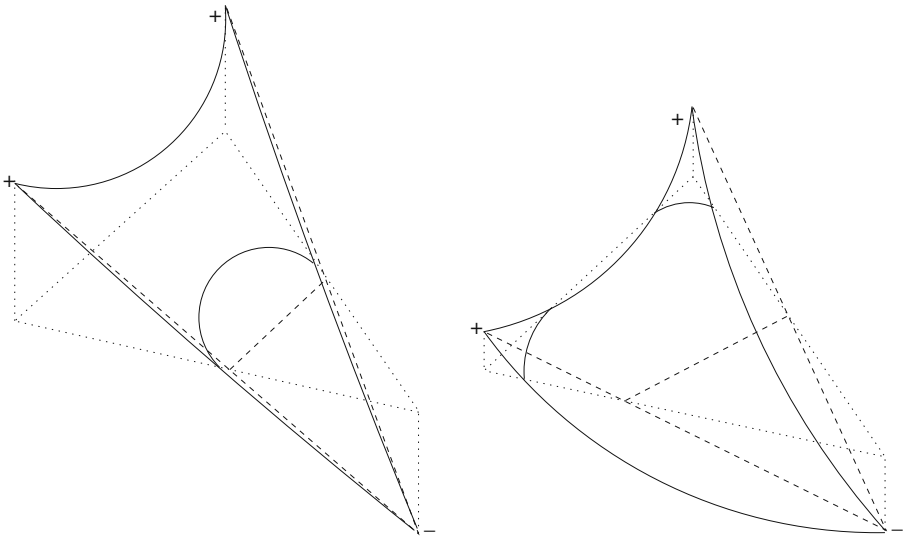

Fig. 8 Examples of poor curvature approximation obtained by linear interpolation

To prevent this, i.e. to better follow the exact segment, we implement in `level` an *adaptive curvature determination* function. This consists in considering an extra mid edge inside the cell (Fig. 9), finding its zero point by the secant method and measuring the height of the triangle given by this zero and the two other ones along the cell edges with sign change. If this height is too large, say greater than a fixed value T_c depending on the final required accuracy (see Section 8 for details), the process is iterated by adding more extra mid edges. In this way the curvature of the segment can be tracked with sufficient accuracy not loosing the good approximation obtained for the zeros. The position of

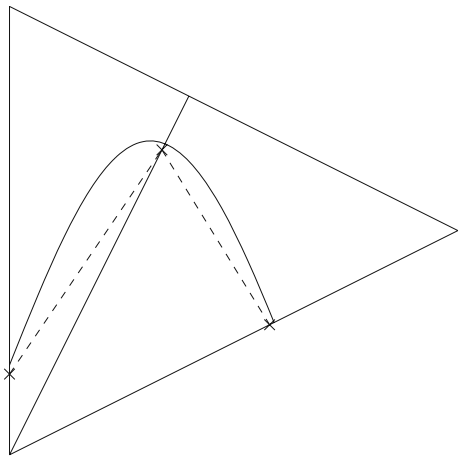

Fig. 9 Extra mid edge for the adaptive curvature determination

each extra mid edge is determined by weighting the position of the zero points along the two edges between which the new one should lie.

In the following section we explain how to overcome the particular case shown in Fig. 8 (right).

7.3 The "two-segments" case

The situation in Fig. 8 (right) could lead to some problem if the extra mid edge computed by the adaptive curvature function falls in between the two segments of level curve. Opposite to the case in Fig. 8 (left), this means that the f vertex values of the new edge have the same sign. In this situation, the secant method along this edge will look for a zero point outside the cell. This would not be a problem if no other level curves were close to the cell, but sometimes this happens and the secant routine determines a zero in the wrong curve.

We avoid this by performing a double secant procedure in order to compute the two zeros which lie on the same edge. Then we start two separate adaptive curvature determinations, one on each external triangle determined with these new points. In this way the two segment are tracked independently and the missing joining part will be given by the neighboring cell.

8 The overall algorithm

In this section we describe the overall implementation of `level` in order to resume how the features presented separately in the previous sections are joint together.

First of all, `level` takes as input the external function f through which the surface values can be computed for any choice of x and y, the constant z identifying the desired set of level curves $f(x, y) = z$, the rectangle $[X_{\min}, X_{\max}] \times [Y_{\min}, Y_{\max}]$ of the (x, y)-plane where the problem has to be solved and the final accuracy TOL on the level curve as defined in Section 4. All the remaining parameters (i.e. the tolerance S_t and the number of iterations S_i for the secant method, the tolerance for the adaptive curvature determination T_c, the tolerances for the slope tests on squares T_s and on triangles T_t and the minimum number p of starting squares along the shortest side) are set to fixed default values given in the sequel. These values have been selected after numerous tests on several functions. Nevertheless, as announced in Section 5, the user is free to modify these values by providing additional optional inputs.

The tolerance for the secant method along the cell edges is set to $S_t = 0.01$, which means that a maximum error of 1% of the edge length is allowed. The maximum number of secant iterations is set to 10. The size of the smallest triangle (i.e. the length of its cathetus) created by the adaptive triangulation is given by

$$l_t = \frac{\mathrm{TOL}}{S_t}.$$

The starting square grid is determined by setting a minimum number of $p = 10$ squares along the shortest edge $\min\{|X_{\max} - X_{\min}|, |Y_{\max} - Y_{\min}|\}$. Then $X_{\max}$ and $Y_{\max}$ are rearranged as explained in Section 6.1 to contain exactly a minimum 10×10 square grid with separation l_s determined accordingly. At this point, if the starting squares are already smaller than the smallest possible triangle, we set automatically $l_t = l_s$, which simply means that the starting grid is accurate enough to match the final tolerance TOL via the secant iterations. If not the case, the adaptive triangulation is performed.

The tolerances T_s and T_t for the slope tests are set to be (possibly different) fixed numbers as said in Section 7.1. From the output of numerous tests on different surfaces, it turns out that the values $T_s = T_t = 0.2$ are large enough for surfaces with "normal" variation, i.e. say with first derivative enough far away from zero, while these values have to be increased when the surface is almost flat. Anyway values T_s, $T_t \leq 5$ should guarantee to find the full set of level curves, even the smaller ones. We point out that higher values of these parameters lead to more evaluations of f.

Finally, the tolerance T_c for the adaptive curvature determination described in Section 7.2 is set to be

$$T_c = \frac{\text{TOL}}{10}.$$

When all the parameters are set, the code starts creating the square cells. Then, following the scanning direction as in Section 6.1, each square is possibly refined according to the presence of sign changes on its vertex f values or to the slope test as described in Section 7.1. If this is the case, the associated matrix S (Section 6) is passed to the refinement function which provides the triangulation and the final segments location as given in Sections 4 and 5. This function starts from the matrix S, where the f values of the four corners are known, by evaluating f at the center point of the square. With this new vertex, four triangular cells are created and each is stored in a 3×2 matrix containing the coordinates (x, y) of each vertex. These four matrices initialize a vector of matrices T of length 4. Then the refinement analysis starts from the last matrix of T and the following two cases are possible:

- if the cell has to be refined according to sign changes or to the slope test, then
  - the matrix corresponding to the originating cell is deleted from T;
  - the subdivision vertex is calculated;
  - f is evaluated by filling the relevant entry in the matrix S;
  - two new triangular cells are created and stored in two new 3×2 matrices added at the end of T.
- if no refinement is required the cell is deleted from T.

The refinement analysis always resume from the last matrix of T and it stops when this vector is empty, that means that the whole region of the (x, y)-plane included in the input square cell represented by S is analyzed.

During the triangulation, when the minimum triangle size is reached and a sign change occur or the slope test detects a possible level curve, the cell is passed to the function implementing the secant method coupled with the adaptive curvature determination as described in Sections 5, 7.2 and 7.3.

Finally, when the refinement of a square is finished, the matrix S of the next square is initialized with the necessary f values given by refinement of the neighboring squares according to what presented in Section 6.1.

A complete description of functions parameters and calls can be found in the primer provided with the relevant software. Moreover, let us remark that the algorithm presented in this paper is used (together with the one of [5]) in "Trace-DDE" ([3, 13]), a Matlab graphic user interface devoted to the computation of characteristic roots and of stability charts of DDEs.

Remark 1 Observe that for small-size closed contours, the size of the starting square grid is also important, i.e. if this latter is larger than the size of such a curve, then the curve might be missed during the algorithm execution. To our knowledge, there is no optimal choice of the initial grid size. Such an optimal choice should be based on accurate estimates of the diameters of the level curves, which are very difficult to obtain. A partial (but effective to our experience on several tests) remedy is represented by the slope test on squares described in Section 7.1 although, as already stated, this does not represent a sufficient condition to exclude the possibility of missing small portions of the set of level curves.

9 Numerical examples

We present here some numerical experiments on the case studies described in Section 2. All presented tests (also in forthcoming sections) are performed on a MacBook Pro 2.53 GHz Intel Core 2 Duo processor with 4 GB 1067 Mhz DDR3 RAM. Similar tests were also performed on a Pentium III processor with 256 MB RAM running Windows XP. Matlab version 7.0 R14 was used.

Functions Test 1 and Test 2 are tested both with `contour` and `level` to compute their set of level curves $f(x, y) = 0$ and $f(x, y) = 3$, respectively. Computational data are collected in Tables 1 and 2 where TOL refers to the final required accuracy given in input, N refers to the number of f evaluations

Table 1 Computational data for Test 1: TOL = accuracy, N = number of f evaluations, t = CPU time (seconds), suffix c for `contour` and l for `level`

TOL	N_c	N_l	t_c	t_l	N_c/N_l	t_c/t_l
0.1	441	522	0.0	0.1	0.8	0.3
0.05	1681	522	0.0	0.2	3.2	0.2
0.01	40401	572	0.1	0.2	70.6	0.5
0.005	160801	614	0.6	0.2	261.9	4.0
0.001	4004001	1007	109.4	0.2	3976.2	454.0

Table 2 Computational data for Test 2: TOL = accuracy, N = number of f evaluations, t = CPU time (seconds), suffix c for `contour` and l for `level`

TOL	N_c	N_l	t_c	t_l	N_c/N_l	t_c/t_l
0.1	3721	789	0.1	0.2	4.7	0.4
0.05	14641	956	0.2	0.2	15.3	0.9
0.01	361121	1727	5.9	0.4	209.1	14.8
0.005	1442401	2241	39.0	0.5	643.6	73.5
0.001		4067		1.0		

needed to calculate and plot the level curves, t refers to the CPU time. The suffix c stands for `contour` and the suffix l stands for `level`. All internal parameters are fixed to the default values as given in Section 8.

From both tables it can be noticed the speedup N_c/N_l in terms of number of f evaluations obtained by using `level` with respect to `contour`. This increases notably as TOL decreases. In particular for Test 2, the value TOL = 0.001 cannot be reached using `contour` due to limited memory capacity: it would require more than 3.6×10^7 evaluations of f against the relatively small amount of 4067 points with `level`.

The same increasing trend occurs in terms of CPU time, but in this case `contour` is still comparable with respect to `level`, except for the lowest values of TOL. This happens because both the functions Test 1 and Test 2 are computed at each required point (x, y) almost instantaneously, hence the major tribute to the computational cost comes from the computational structure of the algorithms more than from the evaluations of f, and the triangulation in `level` is certainly more expensive than the regular grid in `contour`. As stated in the introduction, the advantage in saving computational time of an adaptive strategy turns out to be evident when f is computationally heavy as it will be shown in the next section.

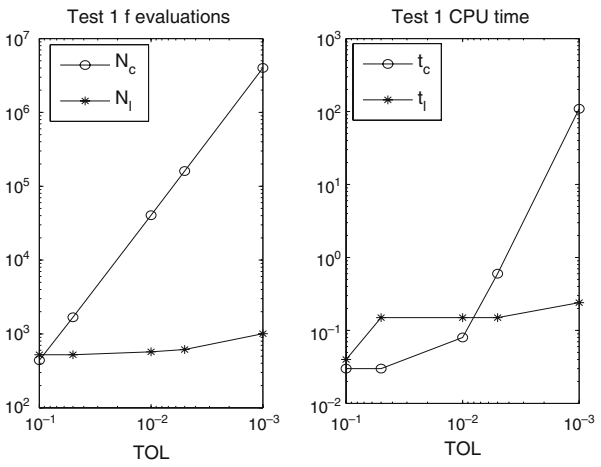

Fig. 10 Computational comparison for Test 1

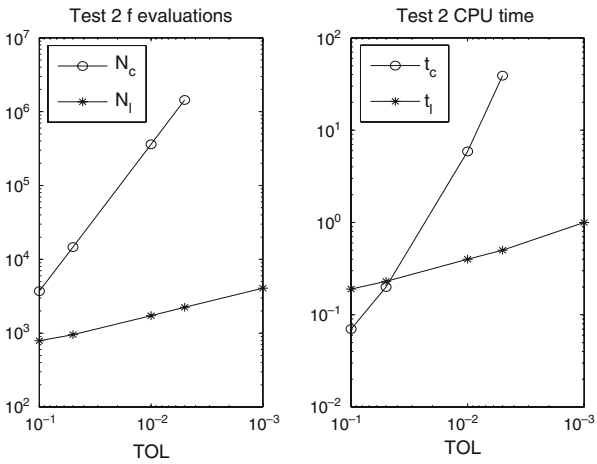

Fig. 11 Computational comparison for Test 2

Figures 10 and 11 resume the computational results by using double logarithmic plots of N and t with respect to TOL for both Test 1 and Test 2.

Finally, Figs. 12 and 13 show the set of level curves at $z = 0$ for Test 1 and at $z = 3$ for Test 2 and the points (with dots) at which f has been evaluated to obtain the curves with $\mathrm{TOL} = 0.05$ and $\mathrm{TOL} = 0.1$, respectively, using `contour` (left) and `level` (right). The use of the adaptive strategy (right) is evident.

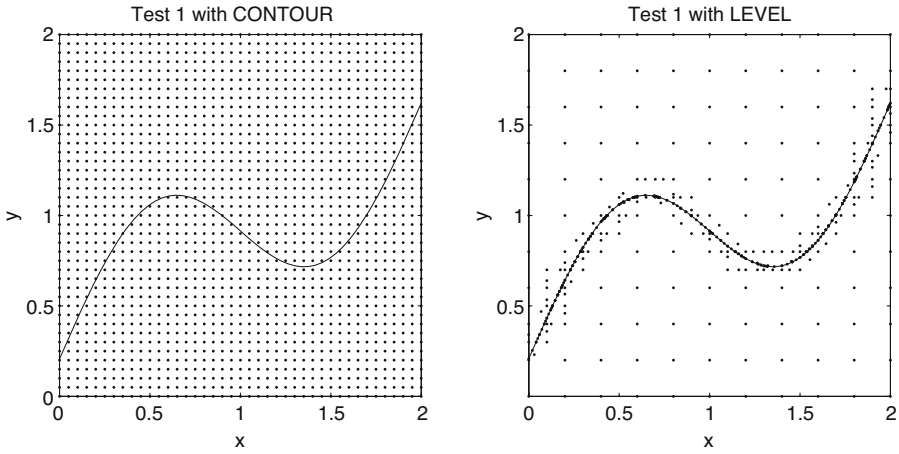

Fig. 12 Evaluations of f (*dots*) for Test 1: 1681 with `contour` (*left*) and 522 with `level` (*right*) for $\mathrm{TOL} = 0.05$

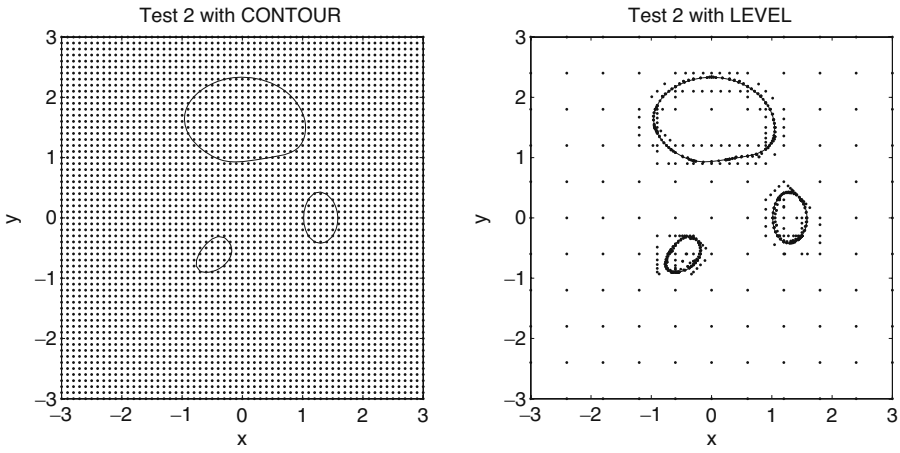

Fig. 13 Evaluations of f (*dots*) for Test 2: 3721 with `contour` (*left*) and 789 with `level` (*right*) for TOL = 0.1

10 Real-life applications

10.1 Stability charts

In this section we face the problem of computing the *stability chart* of a system of linear DDEs of the general form

$$y'(t) = L_0 y(t) + \sum_{l=1}^{k} \left(L_l y(t-\tau_l) + \int_{-\tau_l}^{-\tau_{l-1}} M_l(\theta) y(t+\theta) d\theta \right), \ t \geq 0, \qquad (3)$$

where $L_0, L_1, \ldots, L_k \in \mathbb{C}^{m\times m}$, $0 = \tau_0 < \tau_1 < \cdots < \tau_k = \tau$ and $M_l : [-\tau, 0] \to \mathbb{C}^{m\times m}$, $l = 1, \ldots, k$, are smooth functions. Delay systems such as (3) are particularly important in control theory, where the stability effects of delays are a crucial problem [10, 12]. Important applications can be found also in machining tool such as milling, turning and drilling where the role of parameters such as spindle speed and feed are stability determining [8, 11]: these are second order systems with time dependent coefficients and the interest is in the stability of periodic solutions.

The asymptotic stability of the zero solution of (3) is determined by the position on $\mathbb{C}$ of the rightmost characteristic root, i.e. the solution with largest real part $\lambda_r \in \mathbb{C}$ of the characteristic equation

$$\det \left(\lambda I - L_0 - \sum_{l=1}^{k} \left(L_l e^{-\lambda\tau_l} + \int_{-\tau_l}^{-\tau_{l-1}} M_l(\theta) e^{\lambda\theta} d\theta \right) \right) = 0. \qquad (4)$$

In particular, it is well known that the zero solution is asymptotically stable if and only if $\Re(\lambda_r) < 0$ [7].

Now suppose that system (3) depends on two uncertain parameters (e.g. delays or coefficients) p_1 and p_2 given into fixed intervals, i.e. $p_1 \in [p_{1,\min}, p_{1,\max}]$ and $p_2 \in [p_{2,\min}, p_{2,\max}]$. The *stability chart* is nothing else but the collection of stable-unstable regions in the rectangle $[p_{1,\min}, p_{1,\max}] \times [p_{2,\min}, p_{2,\max}]$ of the parameters plane. Clearly the regions are determined by the so-called *stability boundaries*, i.e. the set of curves $\lambda_r(p_1, p_2) = 0$, being the system asymptotically stable wherever $\lambda_r(p_1, p_2) < 0$ and unstable elsewhere. Indeed this problem corresponds to find the set of $z = 0$ level curves of the surface $f(x, y) = \lambda_r(p_1, p_2)$ in $[X_{\min}, X_{\max}] \times [Y_{\min}, Y_{\max}] = [p_{1,\min}, p_{1,\max}] \times [p_{2,\min}, p_{2,\max}]$ and hence it can be solved by using `contour` or `level` as described in this paper.

This real-life application is a challenging problem. In fact, the characteristic equation (4) is transcendental and the infinitely many characteristic roots cannot be computed analytically, rather a finite set of them can be numerically approximated. In the recent years the authors presented a family of numerical techniques focused on the discretization of the infinitesimal generator of the solution semigroup associated to (3) [2, 4, 5]. The discretization via pseudospectral differencing techniques [5] is based on $n + 1$ Chebyshev nodes on the delay interval $[-\tau, 0]$ and it leads to a matrix whose eigenvalues give approximations to the rightmost characteristic roots. Consequently, each evaluation of the function $\lambda_r(p_1, p_2)$ corresponds to a (possibly large) eigenvalue problem, hence it is computationally expensive and the use of an adaptive strategy with respect to a regular grid in order to plot the stability boundaries reveals itself substantially advantageous in terms of computational time as we show in the following case study.

As an application (indicated as "Test 3" in the sequel and in the relevant software) we consider a case of variable pitch cutter applied in modern machining whose dynamics is modeled with the following system of 8 DDEs with five discrete delays depending on the two parameters τ_1 and τ_2 ([1] and courtesy of Prof. N. Olgac and Dr. R. Sipahi, University of Connecticut, Mechanical Engineering Departement):

$$\begin{aligned} y'(t) = {} & L_0 y(t) + L_1(y(t-\tau_1) + y(t-\tau_2)) + L_2(y(t-2\tau_1) + \\ & + y(t-2\tau_2)) + L_3 y(t-\tau_1-\tau_2). \end{aligned} \tag{5}$$

The associated stability chart for $(\tau_1, \tau_2) \in [0, 2 \times 10^{-3}] \times [0, 2 \times 10^{-3}]$ is depicted in Fig. 14: each evaluation of λ_r is obtained by the use of pseudospectral differencing methods based on $n + 1$ Chebyshev nodes, which means a final eigenvalue problem of dimension $m(n + 1)$ [5] where $m = 8$ is the system dimension. We performed two similar computations with $n = 10$ and $n = 40$, the latter resulting more accurate due to the finer discretization in the numerical procedure for the rightmost root approximation. Indeed lower values of n do not lead to correct boundaries as is the case in Fig. 14 for $n = 10$ (the dimension of the corresponding eigenvalue problem is 88). On the other hand, the larger is n, the more expensive is the computation of the rightmost root for one choice of the two parameters, evaluation which takes around 3 s on average with $n = 40$ (the dimension of the corresponding eigenvalue problem is 328).

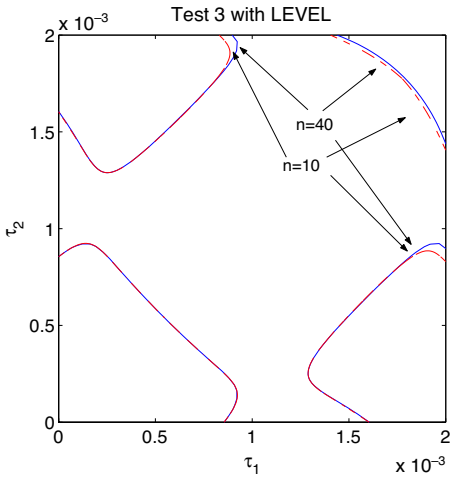

Fig. 14 Stability chart of system Test 3 computed with `level` for TOL $= 1 \times 10^{-5}$, $n = 10$ (*dashed line*) and $n = 40$ (*solid line*), the origin is a stable point

A computational comparison between `contour` and `level` is reported in Table 3 and Fig. 15. It is now clear from these data that the adaptive triangulation strategy implemented in `level` gives raise to a considerable reduction in either the number of function evaluations and the CPU time. In Table 3, for $n = 40$, the last value of N_c $(*)$ is deduced from the regular grid size giving a final accuracy of TOL $= 1 \times 10^{-5}$ and $t_c \simeq 3$ h is estimated from N_c knowing the average cost of 3 s per evaluation: compared to $t_l \simeq 4$ min the advantage is evident.

10.2 $\varepsilon-$pseudospectra

In this section we consider the computation of the $\varepsilon-$pseudospectrum of a linear operator $\mathscr{A} : X \to X$ where X is a Banach space. We denote by $\Lambda(\mathscr{A})$ the spectrum of $\mathscr{A}$, i.e the set of $\lambda \in \mathbb{C}$ such that a bounded inverse of $\lambda I - \mathscr{A}$ does not exist. For any $\varepsilon > 0$, the $\varepsilon-$pseudospectrum $\Lambda_\varepsilon(\mathscr{A})$ of $\mathscr{A}$ is defined as [14]

$$\Lambda_\varepsilon(\mathscr{A}) = \left\{\lambda \in \mathbb{C} : \| (\lambda I - \mathscr{A})^{-1} \|_\infty \geq \varepsilon^{-1}\right\} \tag{6}$$

Table 3 Computational data for Test 3: n = discretization index, TOL = accuracy, N = number of f evaluations, t = CPU time (seconds), suffix c for `contour` and l for `level` (* estimated)

	TOL	N_c	N_l	t_c	t_l	N_c/N_l	t_c/t_l
$n = 10$	1×10^{-4}	441	627	4.4	6.3	0.7	0.7
	5×10^{-5}	1681	667	17.0	6.6	2.5	2.6
	1×10^{-5}	40401	881	390.9	8.9	45.9	43.9
$n = 40$	1×10^{-4}	441	641	117.6	168.6	0.7	0.7
	5×10^{-5}	1681	667	446.9	175.7	2.5	2.5
	1×10^{-5}	40401	915	10499*	240.7	44.1*	43.6*

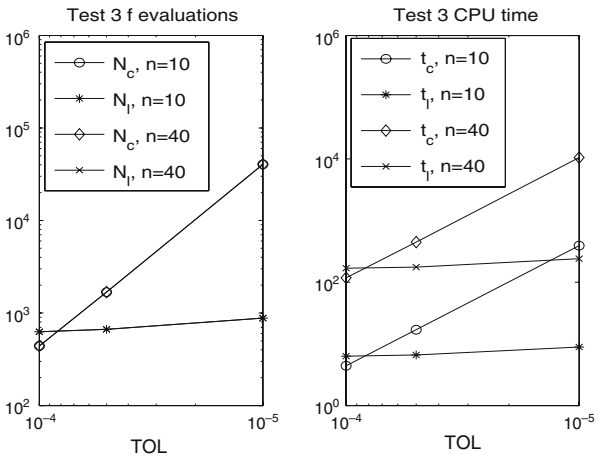

Fig. 15 Computational comparison for Test 3

and assuming by convention that $\|(\lambda I - \mathscr{A})^{-1}\|_\infty = \infty$ for $\lambda \in \Lambda(\mathscr{A})$, it is clear that $\Lambda_0(\mathscr{A}) = \Lambda(\mathscr{A})$, but it can be shown that $\Lambda_\varepsilon(\mathscr{A}) \supset \Lambda(\mathscr{A})$ for $\varepsilon > 0$.

The reason why pseudospectra are important is mainly that in the case of (matrices or) operators far from normality, the knowledge of their spectrum is not always suitable to get information about their behavior measured by quantities such as $\|\exp(t\mathscr{A})\|$ or $\|\mathscr{A}^n\|$, [14]. For instance, in the case of a differential operator $\mathscr{A}$ governing the dynamics of an evolving system, the spectrum of $\mathscr{A}$ can give information about the asymptotic behavior of the zero solution, but nothing can be said about the transient which can exhibit a fast and large growth prior to decay in the case of stability, i.e. eigenvalues in the left-half of $\mathbb{C}$. The analysis of the pseudospectrum can provide such information.

Moreover, the alternative definition

$$\Lambda_\varepsilon(\mathscr{A}) = \{\lambda \in \mathbb{C} : \lambda \in \Lambda(\mathscr{A} + \Delta\mathscr{A}) \text{ for some } \Delta\mathscr{A} \text{ with } \|\Delta\mathscr{A}\| \leq \varepsilon\}$$

suggests that the pseudospectrum measures the perturbation of the spectrum of an operator subject to perturbations itself.

In this work we consider the particular case arising again from DDEs, i.e. we compute the $\varepsilon-$pseudospectrum of the linear unbounded operator $\mathscr{A}$ which is the infinitesimal generator associated to systems of DDEs such as (3) [9]. Since this operator is infinite dimensional, its pseudospectrum is approximated by discretizing $\mathscr{A}$ into a suitable matrix A_n via pseudospectral differencing methods as reported in the previous section, for details see [5]. Although there is little general literature on the computation of pseudospectra of infinite dimensional operator via a matrix discretization, "this procedure can be quite successful if the discretization is highly accurate, and, in particular, spectral methods rather than finite differences or finite elements have been the basis of most the computations so far" [14].

It is clear from (6) that the $\varepsilon-$pseudospectrum is bounded by the level curves $f(x, y) = \varepsilon^{-1}$ of the function

$$f(x, y) = \| (\lambda I - \mathscr{A}_n)^{-1} \|_\infty, \ \lambda = x + \mathrm{i}y, \ \mathrm{i}^2 = -1,$$

where x and y are real and $\mathscr{A}_n$ is the matrix discretization of the infinitesimal generator $\mathscr{A}$ associated to the system of DDEs. Therefore the problem can be solved again either with `contour` or `level`.

In the sequel we report about the computation of the pseudospectra relevant to the DDEs:

$$y'(t) = -5y(t) - y(t-1) \tag{7}$$

and

$$y''(t) = \frac{k^2}{2}(y(t) + y(t-\tau)) \tag{8}$$

with $k = 1$ and $\tau = 4$. Equation (7) (indicated as "Test 4" in the sequel and in the relevant software) is a single delay case used as a constructed test. The second order equation (8) (provided by Prof. K. Bohinc, University of Ljubljana, during a private communication and indicated as "Test 5" in the sequel and in the relevant software) models the potential y of the electric field relevant to a molecule positioned perpendicularly at a distance t from a uniformly charged plane. In this last case the variable t denotes a spatial variable, hence τ is a spatial "delay" and, moreover, the model can be easily reduced to a first order system of two DDEs.

Numerical results are collected in Table 4 and refer to the computation of the level curves at the ten different values $\varepsilon = 10^{-s}$ with $s = -0.15 : 0.1 : 0.75$ for Test 4 and $s = 0.7 : 0.2 : 2.5$ for Test 5. Since `contour` is based on a uniform grid, the same grid points are used to compute all the level curves, hence the computational effort (in terms of number of grid points) does not change for one or ten levels. Opposite, the adaptive strategy adopted in `level` requires one computation for each level because of the dependence of the grid from the level curve itself. Hence, in Table 4 the number of f evaluations refers to the total required for all the 10 levels. The same holds for the CPU time. Although this, it can be noticed how `level` is still advantageous with respect to `contour`, the reason lying in the (large) computational cost of a single f evaluation. It is then clear that the higher is this cost, the better performing is `level`.

Table 4 Computational data for Test 4 (top) and Test 5 (bottom): n = discretization index, TOL = accuracy, N = number of f evaluations, t = CPU time (seconds), suffix c for `contour` and l for `level`

n	TOL	N_c	N_l	t_c	t_l	N_c/N_l	t_c/t_l
20	0.05	20301	9862	4.6	4.5	2.1	1.0
20	0.01	20301	8265	25.6	9.8	2.5	2.6

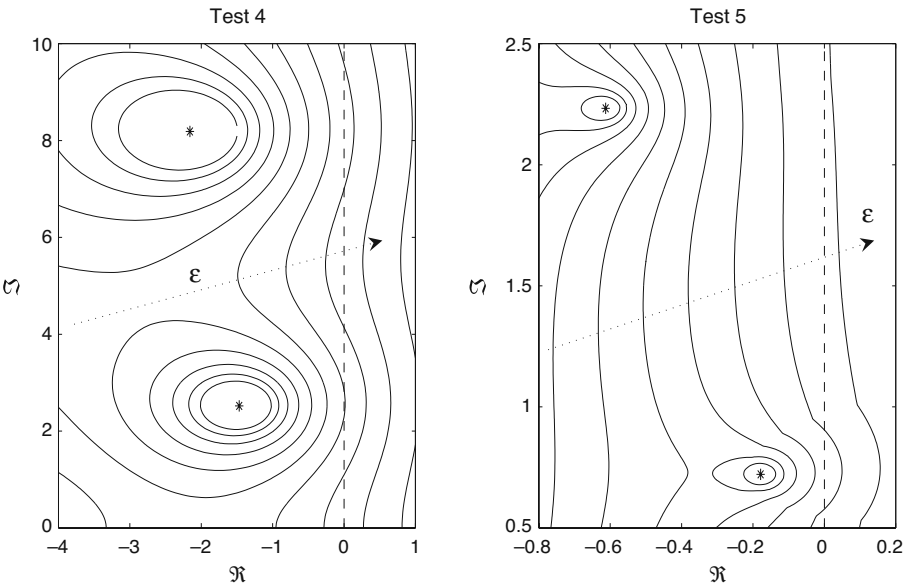

Fig. 16 $\varepsilon-$pseudospectrum for Test 4 (*left*) and Test 5 (*right*), *arrow* denotes increasing ε

Representations of the $\varepsilon-$pseudospectra for Test 4 and Test 5 are shown in Fig. 16 around the second and third rightmost eigenvalues (with positive imaginary part since the spectrum is symmetric with respect to the real axis). The imaginary axis is also depicted: since it represents the limit between asymptotically stable and unstable behavior, its intersection with the $\varepsilon-$pseudospectrum gives information about the minimum amount of perturbation of $\mathscr{A}$ which leads to bifurcation.

References

1. Altintas, Y., Engin, S., Budak, E.: Analytical stability prediction and design of variable pitch cutters. J. Manuf. Sci. E.-T. ASME **121**, 173–178 (1999)
2. Breda, D.: The infinitesimal generator approach for the computation of characteristic roots for delay differential equations using BDF methods. Technical Report RR17/2002, Department of Mathematics and Computer Science, University of Udine, (2002)
3. Breda, D., Maset, S., Sechi, D., Vermiglio, R.: Trace-DDE. http://users.dimi.uniud.it/~dimitri.breda/software.html (2005)
4. Breda, D., Maset, S., Vermiglio, R.: Computing the characteristic roots for delay differential equations. IMA J. Numer. Anal. **24**(1), 1–19 (2004)
5. Breda, D., Maset, S., Vermiglio, R.: Pseudospectral differencing methods for characteristic roots of delay differential equations. SIAM J. Sci. Comput. **27**(2), 482–495 (2005)
6. Dahlquist, G., Björck, Å: Numerical Methods. Prentice-Hall, Englewood Cliffs (1974)
7. Hale, J.K., Verduyn Lunel, S.M.: Introduction to Functional Differential Equations, No. 99, AMS series. Springer, New York (1993)
8. Insperger, T., Stépán, G.: Updated semi-discretization method for periodic delay-differential equations with discrete delay. Int. J. Numer. Methods Eng. **61**, 117–141 (2004)

9. Michiels, W., Green, K., Wagenknecht, T., Niculescu, S.I.: Pseudospectra and stability radii for analytic matrix functions with application to time-delay systems. Linear Algebra Appl. **418**(1), 315–335 (2006)
10. Niculescu, S.I.: Delay Effects on Stability: A Robust Control Approach, No. 269, TLNCIS. Monograph. Springer, London (2001)
11. Olgac, N., Sipahi, R.: An exact method for the stability analysis of time delayed LTI systems. IEEE Trans. Automat. Contr. **47**(5), 793–797 (2002)
12. Richard, J.P.: Time-delay systems: an overview of some recent advances and open problems. Automatica **39**, 1667–1694 (2003)
13. Sechi, D.: Sviluppo di interfaccia grafica per lo studio della stabilità di sistemi differenziali con ritardo. Master's thesis, University of Udine (2005, in italian)
14. Trefethen, L.N.: Pseudospectra of linear operators. SIAM Rev. **39**(3), 383–406 (1997)