Efficient computation of stability charts for linear time delay systems

Dimitri Breda[1] Stefano Maset[2] Rossana Vermiglio[3]

[1]*Dipartimento di Matematica e Informatica*
Università degli Studi di Udine, via delle Scienze 208
I-33100 Udine, Italy
e-mail: `dbreda@dimi.uniud.it`

[2]*Dipartimento di Matematica e Informatica*
Università degli Studi di Trieste, via Valerio 12
I-34127 Trieste, Italy
e-mail: `maset@univ.trieste.it`

[3]*Dipartimento di Matematica e Informatica*
Università degli Studi di Udine, via delle Scienze 208
I-33100 Udine, Italy
e-mail: `vermiglio@dimi.uniud.it`

Research Report UDMI/13/2004/RR

Abstract

A new efficient algorithm for the computation of the stability chart of linear time delay systems is proposed and tested on several examples. The stability chart is obtained by investigating the 2d-parameter space by a first coarse square grid which is then adaptively refined by triangulation to match the desired tolerance. This leads to a considerable reduction in computational cost with respect to investigate a uniform fine square grid. Stability of each point is determined by approximating the rightmost characteristic root real part via a numerical scheme recently developed by the authors and based on pseudospectral differencing methods. A Matlab code is included in appendix.

1 Introduction

Many real phenomena in physics, engineering, chemistry, biology, economics, etc. are better modeled and/or simulated if time delays are taken into account.

All delay systems are characterized by the common feature of being influenced, in their present evolution, by information on their past history. The effects of the presence of delays in the system properties and behavior still feed a wide interest both in research and applications fields. Much of this interest is concerned with the stability analysis of the *linear* case. The lack of good estimates of the parameter values (e.g. delays) involved in system models leads to develop opportune criteria to determine not only whether a nominal system is stable or not, but an entire stability region of parameters due to this uncertainty. When we deal with two varying parameters, we talk about *stability charts*.

In this work we focus on the system of m-dimensional linear delay differential equations (DDEs) with multiple discrete and distributed delays

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

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 (1) are particularly important in control theory, where the stability effects of delays are a crucial problem [Ric03], [Nic00]. 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 [IS04]: these are second order systems with time dependent coefficients and the interest is in the stability of periodic solutions.

It is well known [HVL93] that the zero solution of (1) is asymptotically stable if and only if all the characteristic roots, i.e. the (infinitely many) roots of

$$\det(\Delta(\lambda)) = 0, \tag{2}$$

where

$$\Delta(\lambda) = \lambda I - L_0 - \sum_{l=1}^{k} \left(L_l e^{-\lambda\tau_l} + \int_{-\tau_l}^{-\tau_{l-1}} M_l(\theta) e^{\lambda\theta} d\theta \right), \; \lambda \in \mathbb{C}, \tag{3}$$

have strictly negative real part. Since in every vertical strip there is only a finite number of characteristic roots, the asymptotic stability depends on the sign of the real part of the rightmost characteristic root and we use this fact to determine stability. Numerical methods to compute the rightmost root of (1) are discussed in Section 2.

Once we have a tool to determine the stability of (1) for every choice of its parameters (e.g. coefficients and/or delays), we can proceed to scan the 2d-parameter plane in the following efficient way. First we set a coarse square grid and for each square we determine the stability of its vertices. If they all have the same property, e.g. all “stable” (“unstable”), then the square is set to be stable (unstable) and no refinement is required. Rather, if some vertex is stable and some is not, this means that a portion of stability boundary passes through the square, hence a refinement is required. This further analysis is carried out by evaluating the stability of the center point of the square and dividing it into four triangles. The stability test is then repeated for each of these triangles and if further refinement is needed, then the mid point of the hypotenuse is analyzed and the triangle is divided into two smaller ones. Moreover, a second stability test is done in order to avoid that all the vertices of a cell (square or triangle) have the same stability property but the stability boundary cross an edge of the cell twice. Also in this case a refinement is required. The algorithm goes on until a given size of the cells with different stability property at the vertices is reached.

2 Numerical computation of characteristic roots

In the last few years, numerical approaches for characteristic roots computation have been proposed, which are based on the discretization of either the solution operator associated to (1) or the infinitesimal generator of the solution operator semigroup. We briefly recall that the solution operator $T(t)$, $t \geq 0$, associated to (1) is defined by

$$T(t)\varphi = y_t,\ \varphi \in X,$$

where $X = C\left([-\tau, 0], \mathbb{C}^m\right)$ endowed with the maximum norm, y_t is the function

$$y_t(\theta) = y(t+\theta),\ \theta \in [-\tau, 0],$$

and y is the solution of (1) with initial data $\varphi \in X$. The family $\{T(t)\}_{t\geq 0}$ is a C_0-semigroup with infinitesimal generator $\mathcal{A} : D(\mathcal{A}) \subseteq X \to X$ given by

$$\mathcal{A}\psi = \psi',\ \psi \in D\left(\mathcal{A}\right), \tag{4}$$

with domain

$$D\left(\mathcal{A}\right) = \left\{ \psi \in X \mid \psi' \in X \text{ and } \psi'(0) = L_0\psi(0) + \sum_{l=1}^{k} \left(L_l\psi(-\tau_l) + \right.\right. \tag{5}$$
$$\left.\left. + \int_{-\tau_l}^{-\tau_{l-1}} M_l(\theta)\psi(\theta)d\theta \right) \right\}.$$

So (1) can be restated as the abstract Cauchy problem [DGVLW95]

$$\begin{cases} \frac{dy_t}{dt} = \mathcal{A}y_t,\ t > 0 \\ y_0 = \varphi \end{cases}.$$

The two following important results [DGVLW95], [HVL93]

1. $\det(\Delta(\lambda)) = 0 \Leftrightarrow \lambda = \frac{1}{t}\ln\mu,\ \mu \in \sigma(T(t)) \setminus \{0\}$;
2. $\det(\Delta(\lambda)) = 0 \Leftrightarrow \lambda \in \sigma(\mathcal{A})$;

where $\sigma(\cdot)$ denotes the spectrum, suggest the idea to turn the characteristic roots approximation problem into a corresponding eigenvalue problem for suitable matrix discretization of either $T(t)$ (i.e. *solution operator* approach) or $\mathcal{A}$ (i.e. *infinitesimal generator* approach).

Engelborghs and Roose propose in [ER02] and [ER99] the solution operator approach via linear multistep (LMS) time integration for system (1) without distributed delay term. Their method computes approximations to the roots from a large, standard and sparse eigenvalue problem and it is implemented in the MATLAB package DDE-BIFTOOL for DDEs bifurcation analysis [ELR02], [ELS01]. The distributed delay case is considered in [LER03] by using LMS methods and in [Bre04] by using Runge-Kutta (RK) methods. The complete development of the infinitesimal generator approach first appears in [BMV04c], [Bre02], [BMV04a] where a matrix approximation to $\mathcal{A}$ is obtained discretizing the derivative in (4) by RK, LMS and pseudospectral differencing methods, respectively. The last technique involves the exact differentiation of interpolants at selected sets of points. The resulting differentiation matrix is nonsparse, but we can take advantage of the well-known "spectral accuracy" to obtain very accurate approximation with small matrix dimension. This behavior represents in fact, for sufficiently small tolerance, the outstanding advantage of this method compared to the previously cited discretization schemes. Therefore we choose this one as the core algorithm for determining the stability of each point of the 2d-parameter space and we refer the interested readers to [BMV04a] for further details on convergence and implementation.

Pseudospectral differentiation can be applied even to more general classes of linear functional differential systems in order to numerically compute the (stability determining) eigenvalues of related derivative operators with nonlocal boundary conditions [BMV04b] such as the infinitesimal generator for the DDEs case. Examples are neutral DDEs, age-structured population dynamics governed by integral equations, mixed-type (advanced and retarded) functional differential equations and partial differential equations with delay. Stability of periodic solutions of second order DDEs with time dependent coefficients can be determined by approximating the dominant characteristic multipliers (i.e. the eigenvalues of the solution operator semigroup) and pseudospectral techniques apply as well. No matter what the system type is, when we have a numerical technique which provide us with some stability information about a certain choice of the system parameters, the algorithm for the computation of the stability chart remains unchanged.

3 Stability charts computation

In this section we present the algorithm for computing the stability charts. In particular we start in Section 3.1 describing the conditions that requires a deeper stability analysis of a "cell", i.e. a portion of the 2d-parameter plane. Then we follow in Section 3.2 analyzing how these cells are refined. In Section 3.3 we talk about the initial coarse grid from which the algorithm begins the automatic detection of the stability boundaries. Finally we end in Section 3.4 discussing about how representing the output.

3.1 Refinement tests

Consider a triangular or square cell in the 2d-parameter plane with shortest edge l and suppose that the core algorithm for the numerical computation of the rightmost root provide us with the real part r of the rightmost root of (1) for the choices of parameters corresponding to all the vertices of the cell. Hence we know if each vertex is either "stable", i.e. $r < 0$, or "unstable", i.e. $r \geq 0$.

Three tests are carried out in order to decide if a further stability analysis inside this cell is necessary or not. This analysis is what we call the "cell refinement" and it means that the cell is divided into smaller ones which are analyzed in the same way until the conditions for no further refinement are matched. These conditions depend either on the cell size and/or on the stability information about its vertices.

The first test concerns the size of the cell: only if

$$\mathrm{TEST1}: \quad l > \mathrm{TOL},$$

where TOL is the desired tolerance on the resolution of the stability boundary, then the cell might be possibly refined according to the conditions described below.

The second test concerns the stability of the n_v vertices of the cell. If the stability property (i.e. $\mathrm{sign}(r) = r/|r|$) is the same among all the vertices:

$$\mathrm{TEST2}: \quad \mathrm{sign}(r_i) = \mathrm{sign}(r_j) \text{ for all } i, j = 1, \ldots, n_v,$$

then the cell might be possibly refined only according to the third test described below. Rather, if there is at least one change among the vertices, this ensures that a portion of stability boundary is crossing the cell on at least two edges (Figure 1 left) and the cell is refined. This is a sort of two dimensional bisection strategy.

The previous test does not exclude the following situation: r might have the same sign in all the vertices of the cell, but a portion of stability boundary can cross it at only one edge (Figure 1 right). Also in this case a cell refinement is necessary, but the question is how to detect this possibility. In order to do this consider an

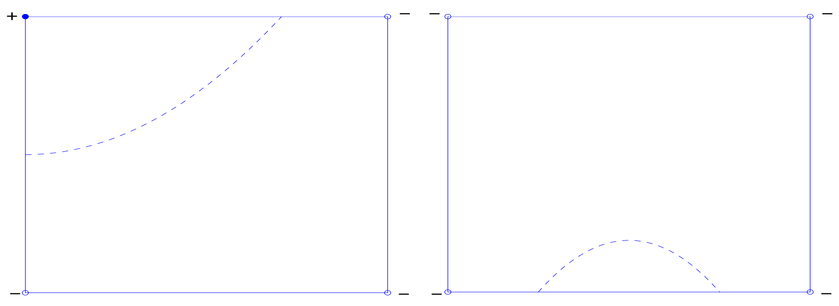

Figure 1: Example of stability boundary (dashed line) crossing a square cell at two edges (left: r changes sign at the vertices) and at one edge (right: r does not change sign at the vertices).

edge $\overline{x_1 x_2}$ with values r_1 and r_2 of the same sign at the vertices (Figure 2). We check the possibility that a stability boundary crossing exists, i.e. there exists a point $x \in \overline{x_1 x_2}$ with $r = 0$, measuring the minimum slope at which x is reached both from r_1 and r_2. This slope is given by

$$s = \tan \alpha = \frac{|r_1 + r_2|}{l}$$

where l is the length of the edge. Then we set a tolerance parameter $\mathrm{TOL_s}$ and if

$$\mathrm{TEST3}: \quad s \geq \mathrm{TOL_s},$$

for all the edges of the cell, then there is no need of refinement because the values of r at the vertices of the edge are “too far” from zero with respect to the length of the edge. Of course this is not a sufficient condition to exclude the refinement, but at least it is a good indicator if $\mathrm{TOL_s}$ is chosen correctly.

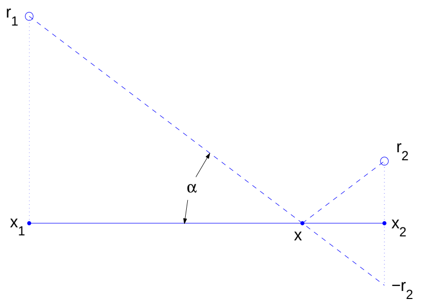

Figure 2: Slope test on a cell edge.

The tests work as follows. If

- TEST2 and TEST3 are true and $r < 0$ for each vertex: the cell is "stable";
- TEST2 and TEST3 are true and $r \geq 0$ for each vertex: the cell is "unstable";
- either TEST2 or TEST3 are false then if
  - TEST1 is false, i.e. the cell size is smaller than TOL, then the cell is "boundary", i.e. it might contains a portion of stability boundary;
  - TEST1 is true, i.e. the cell size is larger than TOL, then the cell is refined.

Among all the tests above, the first is the dominant one, in the sense that if a cell refinement is required according to either the second or the third test, but the cell size is smaller than TOL, then no refinement is done.

3.2 Cell refinement

We start from a square cell of size l_s which has to be refined according to the tests discussed in the previous section. This square cell is first divided by its diagonals into four (isosceles) triangular cells with cathetus l_t, and the stability of the center point is evaluated. Then, if one of these four new triangular cells satisfies the refinement tests, it is divided by its height relevant to the hypotenuse into two (isosceles) triangular cells and the stability of the new vertex is evaluated. The algorithm proceeds in the same way until all the cells match the conditions for no refinement given in the previous section.

The maximum number of possible subdivision is the minimum integer n such that

$$\frac{l_s}{(\sqrt{2})^n} \leq \mathrm{TOL}, \text{ i.e. } \mathrm{n} = \left\lceil 2\log_2\left(\frac{\mathrm{l_s}}{\mathrm{TOL}}\right)\right\rceil$$

where $\lceil p \rceil$ denotes the smallest integer q such that $q \geq p$.

Observe (Figure 3) that each possible new vertex belongs to a $d \times d$ uniform grid of equi-spaced points with separation l_g where

$$d = 2^m + 1, \quad l_g = \frac{l_s}{2^m}$$

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) in the 2d-parameter plane and rightmost root real part r, the corresponding matrix entry is $s_{ij} = r$ with

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

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 stability information can be recovered from the matrix S whenever this vertex is already used in a neighboring cell previously analyzed. For instance in Figure 3, the subdivision of the cell T_1 does not require the evaluation of the stability of the subdivision vertex ($\circ$) 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 stability property, the matrix S is usually sparse and therefore its storing is cheap.

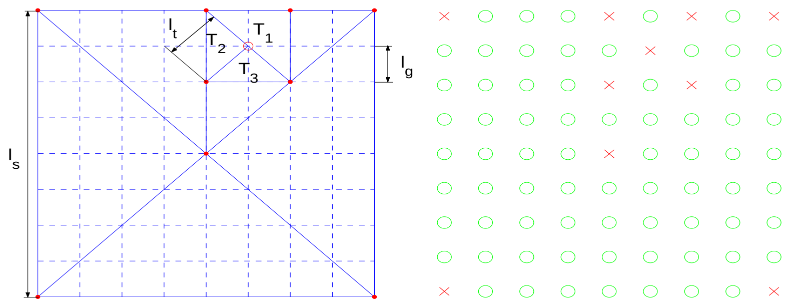

Figure 3: Example of square cell subdivision (left) and its matrix representation (right).

The refinement of a square cell is implemented in a subroutine which starts from the matrix S, where the stability of the four corners is known, by evaluating the stability of the center point of the square. With this new vertex, four triangular cells are created. Every triangular cell 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.

Case 1 (Refinement) *If the cell has to be refined, then*

- *the matrix corresponding to the originating cell is deleted from T;*
- *the subdivision vertex is calculated;*
- *its stability is evaluated by filling the relevant entry in the matrix S;*

- *two new triangular cells are created and stored in two new matrices added at the end of T.*

Case 2 (No refinement) *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 2d-parameter plane included in the input square cell represented by S is analyzed.

3.3 Starting square grid

The algorithm starts by defining a first coarse square grid on the whole 2d-parameter plane. The whole plane is identified by its minimum and maximum (x, y) coordinates X_{min}, X_{max}, Y_{min} and Y_{max}. The size of the grid square is set to be a multiple of the tolerance parameter TOL by an integer p given in input: $l_s = p \cdot \mathrm{TOL}$. To cover all the plane with an integer number of squares we 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 plane, respectively. The stability analysis starts from the left-top square cell and goes on towards the right and bottom directions, i.e. the usual reading/writing ones (Figure 4).

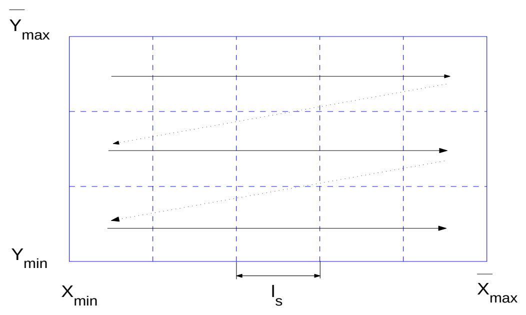

Figure 4: Example of starting square grid on the whole 2d-parameter plane.

Once a square cell has to be refined (the "current" cell, S_c in Figure 5), its information are stored into the matrix S which is passed in input to the refinement subroutine as explained in Section 3.2. As for the triangular 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 Figure 5), respectively "bottom" (S_b in Figure 5). But there might be more vertices in common originated by the refinement. Hence, to avoid any kind of possible multiple stability 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 subroutine 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.

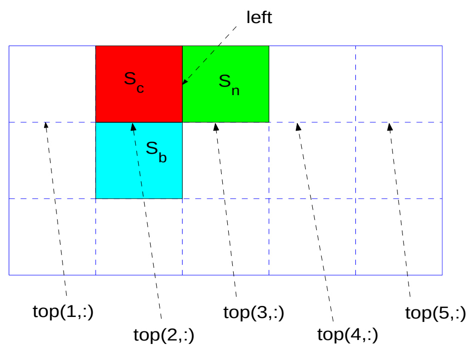

Figure 5: Example of starting square grid on the whole 2d-parameter plane.

3.4 Output representation

There are several ways to represent the output af a stability chart computed as previously described. They depend either on what the user is interested into and on how accurate the representation is asked to be.

For instance, if one is interested in the qualitative behavior of the real part of the rightmost characteristic root r as the two concerned parameters p_1 and p_2 vary, then we can interpret the stability chart as the $\mathbb{R}^2 \to \mathbb{R}$ surface $r = r(p_1, p_2)$ and represent its evolution over the (p_1, p_2)-plane by using a suitable colour mapping varying according to the values of r. In order to get this surface representation it is enough to plot every triangular or square cell by interpolating the colours of its vertices. This is readily obtained by using Matlab's function *fill3* as reported in the code given in Appendix A. Clearly this representation is less accurate far away from the surface zero level.

Opposite, if one is interested only in knowing the stability boundaries, i.e. the curves $r(p_1, p_2) = 0$ in the (p_1, p_2)-plane, then these are to be computed from the information stored in the vertices of each triangular boundary cell. Many strategies can be adopted, the simplest one being linear interpolation of the values of r at the vertices of each cell edge which exhibits a change in the sign of r itself. This is for instance the method used in Matlab's function *contour* for plotting the level curves of a surface. Matlab's *contour* plot is not efficient since it works on a uniform square grid, hence a considerable amount of stability evaluations is required uniformly all over the region. In the following we give a (not detailed) description of a new strategy which leads to accurate stability boundaries requiring a minor number of cells, i.e. a very coarse (and cheap) triangulation. The key idea is to use the secant method on each edge in order to accurately compute the zero of r along that edge. Few iterations are needed for each edge. This is done for the two cell edges which show sign change in r at its vertices. Moreover, a new intermediate edge is computed and its zero is found. This gives an extra boundary point inside the cell. If the three zero points are not sufficiently aligned, it means that the boundary curvature is large. In this case further edges are introduced accordingly, their zeros computed and the boundary line can be drawn in an adaptive way with respect to its curvature.

Results presented in the next section are obtained with both methods described above.

4 Numerical examples

In the following we present the stability charts of the DDEs listed below computed as described in this paper with the Matlab code given in Appendix A. For each test we give both the output representations as discussed in Section 3.4, i.e. the surface corresponding to the real part of the rightmost characteristic root as a function of the two varying parameters (left figures) and the stability boundaries in the $2d$-parameter plane computed with the secant method and the adaptive curvature determination (right figures).

Example 3 *Single delay equation with varying parameters a and b:*

$$y'(t) = ay(t) + by(t-1)$$

Example 4 *$2d$-system with two discrete delays τ_1 and τ_2 as parameters [SO04]:*

$$y'(t) = \begin{pmatrix} -6.45 & -12.1 \\ 1.5 & -0.45 \end{pmatrix} y(t) + \begin{pmatrix} -6 & 0 \\ 1 & 0 \end{pmatrix} y(t-\tau_1) + \begin{pmatrix} 0 & 4 \\ 0 & -2 \end{pmatrix} y(t-\tau_2)$$

Example 5 *$2d$-system with one discrete delay, two distributed delays and varying parameters τ_1 and τ_2 [FSD00]:*

$$y'(t) = L_0 y(t) + L_1 y(t-1) + \int_{-\tau_1}^{-0.1} M_1 y(t+\theta)d\theta + \int_{-\tau_2}^{-0.5} M_2 y(t+\theta)d\theta$$

with coefficients matrices

$$L_0 = \begin{pmatrix} -3 & 1 \\ -24.646 & -35.430 \end{pmatrix}, \quad L_1 = \begin{pmatrix} 1 & 0 \\ 2.35553 & 2.00365 \end{pmatrix},$$

$$M_1 = \begin{pmatrix} 2 & 2.5 \\ 0 & -0.5 \end{pmatrix}, \quad M_2 = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$$

Example 6 *$8d$-system with five discrete delays depending on only two parameters τ_1 and τ_2 ([AEB99] and courtesy of Prof. N. Olgac and Dr. R. Sipahi, University of Connecticut, Mechanical Engineering Departement):*

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

Example 7 *Delayed damped Mathieu equation with $k = 0, 0.1, 0.2$, $\varepsilon = 0$ and varying parameters δ and b [IS04]:*

$$y''(t) + ky'(t) + (\delta + \varepsilon \cos 2\pi t/T)y(t) = by(t-2\pi)$$

Table 1 list the input used for the computation of each test. In particular N is the size of the matrix discretizing the infinitesimal generator to approximate the rightmost characteristic root while TOL, T_s, T_t, p, X_{min}, X_{max}, Y_{min} and Y_{max} are explained in Appendix A. S_{tol} is the tolerance for the secant method and corresponds to a maximum error $S_{tol} \cdot TOL$ in the $2d$-parameter plane. S_{iter} is the maximum number of secant method iterations allowed for each edge. Finally T_c is the tolerance for the adaptive curvature determination and it approximately measures the error in the alignment of three consecutive computed points on the stability boundary. In the same table we report also the total number of stability evaluations $eval$ and the total computational time t (on a 550Mhz Pentium 3 processor).

	Example 3		Example 4		Example 5		Example 6		Example 7	
	s	b	s	b	s	b	s	b	s	b
N	10	15	15	30	15	15	40	40	15	15
TOL	0.05	0.5	0.05	0.05	0.5	1	5×10^{-5}	5×10^{-4}	0.05	0.1
T_s	1.2	1.5	1	1	0.2	0.2	10	1	1	1
T_t	1.2	1.5	1	2	0.2	0.2	1.2	1	0.5	1
p	8	2	10	20	2.5	2.5	4	1	20	5
X_{min}	-2		1.5	0	0		0		-1	
X_{max}	2		3.5	5	10		6×10^{-3}		5	
Y_{min}	-2		3	0	0		0		-1.5	
Y_{max}	2		5	5	10		6×10^{-3}		1.5	
S_{tol}	-	0.05	-	0.05	-	0.1	-	0.025	-	0.05
S_{iter}	-	10	-	10	-	10	-	10	-	10
T_c	-	0.01	-	0.01	-	0.1	-	5×10^{-5}	-	0.01
$eval$	1060	262	2910	38513	624	312	4465	4806	2758	3268
t (sec)	14	3	79	2093	128	61	13185	14502	58	56

Table 1: Computation input and output (s=surface, b=boundary).

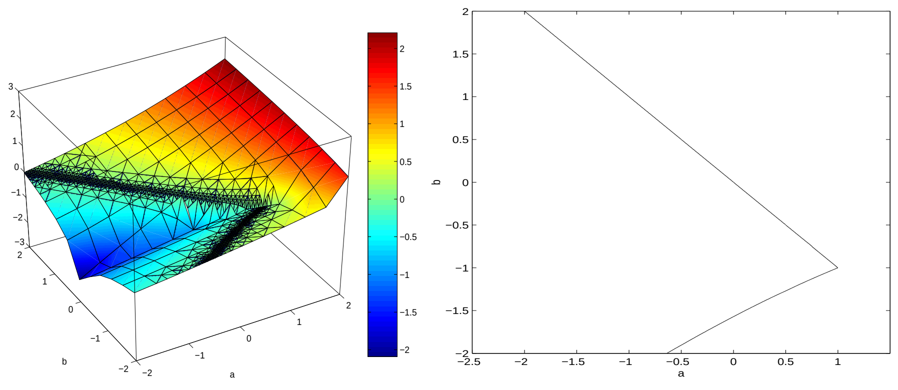

Figure 6: Rightmost root real part surface (left) and stability boundaries (right) on the 2d-parameter plane for example 3.

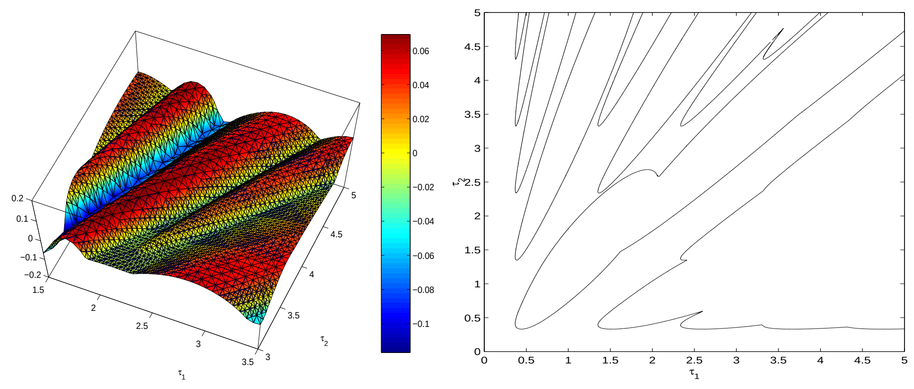

Figure 7: Rightmost root real part surface (left) and stability boundaries (right) on the 2d-parameter plane for example 4.

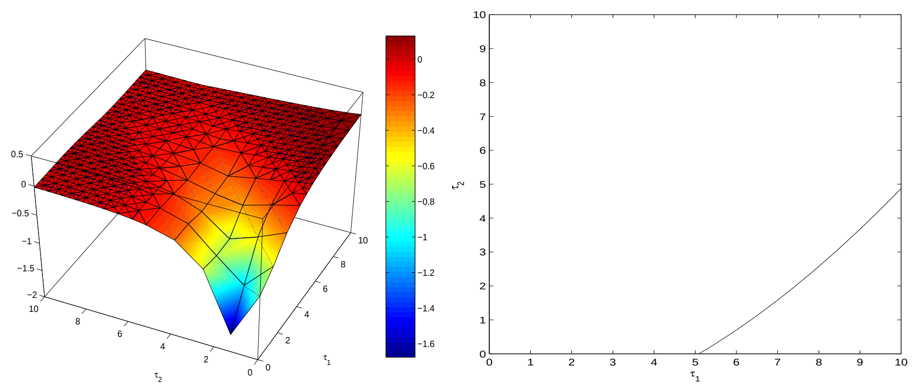

Figure 8: Rightmost root real part surface (left) and stability boundaries (right) on the 2d-parameter plane for example 5.

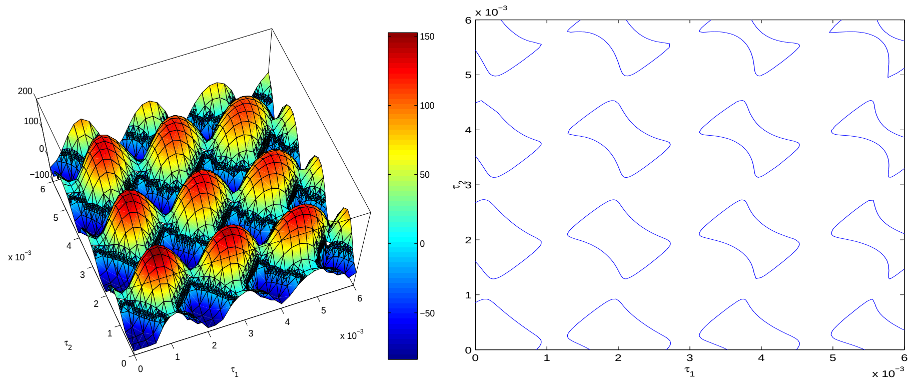

Figure 9: Rightmost root real part surface (left) and stability boundaries (right) on the 2d-parameter plane for example 6.

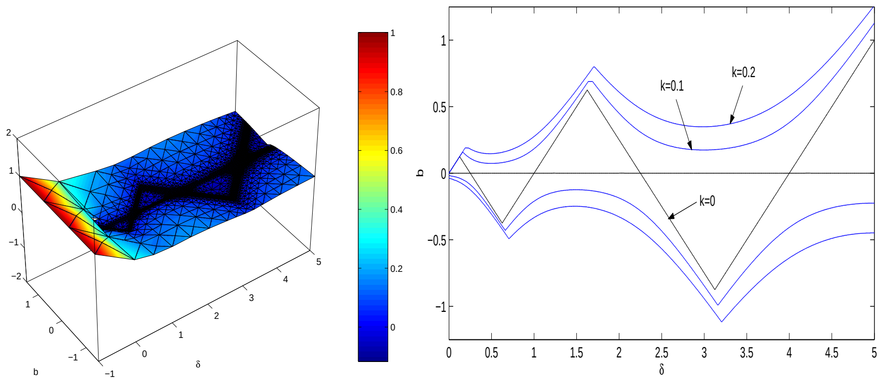

Figure 10: Rightmost root real part surface (left) and stability boundaries (right) on the 2d-parameter plane for example 7.

APPENDIX A: MATLAB code

```
function [sv,tv]=schart(TOL,Ts,Tt,p,Xmin,Xmax,Ymin,Ymax)

%sv=nr of square cell vertices stability evaluations
%tv=nr of triangular cell vertices stability evaluations
%TOL=tolerance parameter on stability boundary thickness
%Ts=tolerance parameter for test3 on square cells
%Tt=tolerance parameter for test3 on triangular cells
%p=integer ratio between TOL and square cell edge length
%Xmin,Xmax,Ymin,Ymax vertices of the 2d-parameter plane
%-------------------------------------------------------
%uses external function ’r=sval(x,y)’ to evaluate the
%rightmost root real part ’r’ of the given DDE for the
%parameters ’x’ and ’y’

hold on                         %figure setting:
%axis equal                     %optional depending on
%axis([Xmin Xmax,Xmin Xmax])    %output style (2d,3d,etc.)

ls=p*TOL;                       %square cell edge length
x=Xmin:ls:Xmax;                 %starting grid x-coordinates
y=Ymax:-ls:Ymin;                %starting grid y-coordinates
nx=length(x)-1;                 %nr of horizontal square cells
ny=length(y)-1;                 %nr of vertical square cells

m=ceil(log2(sqrt(2)*ls/TOL));     %subdivision parameter
d=2ˆm+1;                          %size of matrix ’S’
lg=ls/(d-1);                      %square subdivision edge
```

```
top=sparse(nx,d);                %initialization of the
top(1,1)=sval(Xmin,Ymax);        %auxiliary matrix ’top’
top(nx,d)=sval(Xmax,Ymax);       %on the first row of the
for i=1:nx-1                     %starting square grid
    top(i,d)=sval(x(i+1),Ymax);
    top(i+1,1)=top(i,d);
end

left=sparse(d-1,1);              %initialization of the
left(d-1,1)=sval(Xmin,y(2));     %auxiliary vector ’left’

S=sparse(d,d);                   %initialization of the
S(1,:)=top(1,:);                 %matrix ’S’ representing
S(2:d,1)=left;                   %the first square cell
S(d,d)=sval(x(2),y(2));

sv=nx+3;           %initialize
tv=0;              %initialize
for j=1:ny         %scan starting square grid top to bottom
    for i=1:nx     %scan starting square grid left to right
                   %test square cell for refinement
        [left,trow,v]=ref(TOL,Ts,Tt,S,m,d,lg,ls,x(i),y(j));
        tv=tv+v;         %update
        top(i,:)=trow;   %update row of matrix ’top’
        S=sparse(d,d);                  %update matrix ’S’:
        if i<nx                         %check if the next
            S(1,:)=top(i+1,:);          %square cell is at
            S(:,1)=left;                %the beginning of a
            S(d,d)=sval(x(i+2),y(j+1)); %new row of the
            sv=sv+1;                    %starting grid
        elseif j<ny                     %or if the next
            S(1,:)=top(1,:);            %square cell is the
            S(d,1)=sval(Xmin,y(j+2));   %right-bottom one
            S(d,d)=sval(x(2),y(j+2));   %of the starting
            sv=sv+2;                    %grid
        end
    end
end
%----------------------------------------------------------%
%                  refinement subroutine                   %
%----------------------------------------------------------%
function [right,btm,v]=ref(TOL,Ts,Tt,S,m,d,lg,ls,xmin,ymax)
```

```
x=xmin:lg:xmin+ls;        %x-coordinates of the square cell
y=ymax:-lg:ymax-ls;       %y-coordinates of the square cell
v=0;                      %initialize stability evaluations

if (sign(S(1,1))==sign(S(1,d)))&(sign(S(1,d))==...   %test2
sign(S(d,d)))&(sign(S(d,d))==sign(S(d,1)))&...
    (min(abs([S(1,1)+S(1,d),S(1,d)+S(d,d),...        %test3
S(d,d)+S(d,1),S(d,1)+S(1,1)]))>ls*Ts)
    %square cell output: optional (2d,3d,etc.)
    fill3([x(1),x(d),x(d),x(1)],[y(1),y(1),y(d),y(d)],...
    [S(1,1),S(1,d),S(d,d),S(d,1)],...
    [S(1,1),S(1,d),S(d,d),S(d,1)]);
else c=d/2+.5;                  %square center coordinate
    S(c,c)=sval(x(c),y(c));    %center stability evaluation
    v=v+1;                      %update
    T=zeros(3,2,2*m+2);         %initialize vector 'T'
    T(:,:,1)=[1,1;1,d;c,c];    %of (3x3)-matrices
    T(:,:,2)=[1,d;d,d;c,c];    %representing the first
    T(:,:,3)=[d,d;d,1;c,c];    %four triangular
    T(:,:,4)=[d,1;1,1;c,c];    %cells
    lenT=4;                     %initialize length of 'T'
    while lenT>0                %check if 'T' is empty
        Tc=T(:,:,lenT);         %set current triangular cell
        %compute lenght of triangular cell edges
        lT=sqrt([(Tc(1,1)-Tc(2,1))^2+(Tc(1,2)-Tc(2,2))^2,...
        (Tc(2,1)-Tc(3,1))^2+(Tc(2,2)-Tc(3,2))^2,...
        (Tc(3,1)-Tc(1,1))^2+(Tc(3,2)-Tc(1,2))^2])*lg;
        [maxlT,i]=max(lT);       %compute largest edge
        a=S(Tc(1,1),Tc(1,2));    %recover stability
        b=S(Tc(2,1),Tc(2,2));    %information of the
        c=S(Tc(3,1),Tc(3,2));    %vertices
        if maxlT>TOL                          %test1
            if (sign(a)==sign(b))&...         %test2
                (sign(b)==sign(c))&...
                (min(abs([(a+b)/lT(1),...     %test3
                (b+c)/lT(2),...
                (c+a)/lT(3)]))>Tt)
                %triangular cell output: opt. (2d,3d,etc.)
                fill3(x(Tc(:,2)),y(Tc(:,1)),[a;b;c],[a;b;c])
                T(:,:,lenT)=[];  %update tail of 'T'
                lenT=lenT-1;     %update length of 'T'
            else T1=Tc;          %triangular cell subdiv'n
```

```
                T2=Tc;
                j=mod(i,3)+1;
                xs=(Tc(j,2)+Tc(i,2))/2; %x subdiv'n vertex
                ys=(Tc(j,1)+Tc(i,1))/2; %y subdiv'n vertex
                %subdiv'n vertex stability evaluation only
                %if not previously computed for other cells
                if S(ys,xs)==0
                    S(ys,xs)=sval(x(xs),y(ys));
                    v=v+1;      %update
                end
                T1(i,:)=[ys,xs];
                T2(j,:)=[ys,xs];
                T(:,:,lenT)=T1;  %update tail of 'T'
                lenT=lenT+1;     %update length of 'T'
                T(:,:,lenT)=T2;  %update tail of 'T'
            end
                                 %triangular cell output
        else fill3(x(Tc(:,2)),y(Tc(:,1)),[a;b;c],[a;b;c])
            T(:,:,lenT)=[];      %update tail of 'T'
            lenT=lenT-1;         %update length of 'T'
        end
        Tc=[];                   %empty current cell
   end
end
right=S(:,d);        %update right vector for 'left'
btm=S(d,:);          %update bottom vector for 'top' row
```

References

[AEB99] Altintas, Y., Engin, S. and Budak, E. (1999) *Analytical Stability Prediction and Design of Variable Pitch Cutters*, ASME Journal of Manufacturing Science and Engineering, **121**, 173-178.

[Bre04] Breda, D. (2004) *Solution operator approximation for delay differential equation characteristic roots computation via Runge-Kutta methods*, to appear on special issue of Appl. Numer. Math. for the Third Conference on Volterra and Delay Equations dedicated to Professor Alan Feldstein's 70th Birthday.

[Bre02] Breda, D. (2002) *The infinitesimal generator approach for the computation of characteristic roots for delay differential equations using BDF* *methods*, Research Report RR02/17UDMI, Department of Mathematics and Computer Science, University of Udine, Italy.

[BMV04a] Breda, D., Maset, S. and Vermiglio, R. (2004) *Pseudospectral differencing methods for characteristic roots of delay differential equations*, to appear on SIAM J. Sci. Comput.

[BMV04b] Breda, D., Maset, S. and Vermiglio, R. (2004) *Pseudospectral approximation of eigenvalues of derivative operators with non-local boundary conditions*, to appear on special issue of Appl. Numer. Math. for the Third Conference on Volterra and Delay Equations dedicated to Professor Alan Feldstein's 70th Birthday.

[BMV04c] Breda, D., Maset, S. and Vermiglio, R. (2004) *Computing the characteristic roots for delay differential equations*, IMA J. Numer. Anal., **24** (1), 1-19.

[DGVLW95] Diekmann, O., van Gils, S.A., Verduyn Lunel, S.M. and Walther, H.O. (1995) *Delay Equations - Functional, Complex and Nonlinear Analysis*, Springer Verlag, AMS series n. 110, New York, U.S.A..

[ELR02] Engelborghs, K., Luzyanina T. and Roose, D. (2002) *Numerical bifurcation analysis of delay differential equations using DDE-BIFTOOL*, ACM Trans. Math. Softw., **28** (1), 1-21.

[ELS01] Engelborghs K., Luzyanina T., Samaey G. (2001) *DDE-BIFTOOL v. 2.00: a Matlab package for bifurcation analysis of delay differential equations*, Report TW330, Department of Computer Science, K. U. Leuven, Belgium.

[ER02] Engelborghs, K. and Roose, D. (2002) *On Stability of LMS methods and Characteristic Roots of Delay Differential Equations*, SIAM J. Numer. Anal., **40** (2), 629-650.

[ER99] Engelborghs, K. and Roose, D. (1999) *Numerical computation of stability and detection of Hopf bifurcations of steady-state solutions of delay differential equations*, Advances in Computational Mathematics, **10** (3-4), 271-289.

[FSD00] Fattouh, A., Sename, O. and Dion, J.M. (2000) H_∞ *controller and observer design for linear systems with point and distributed delays*, proceedings of 2nd IFAC workshop on Linear Time Delay Systems, 11-13 September 2000, Ancona, Italy.

[HVL93] Hale, J.K. and Verduyn Lunel, S.M. (1993) *Introduction to functional differential equations*, Springer-Verlag, AMS series n. 99, New York, U.S.A..

[IS04] Inseperger T. and Stépán G. (2004) *Updated semi-discretization method for periodic delay-differential equations with discrete delay*, Int. J. Numer. Meth. Engng, **61**, 117-141.

[LER03] Luzyanina T., Engelborghs, K. and Roose, D. (2003) *Computing stability of differential equations with bounded distributed delays*, Numer. Algorithms, **34** (1), 41-66.

[Nic00] Niculescu, S.I. (2000) *Delay Effects on Stability: A Robust Control Approach*, Monograph Springer, LNCIS n.269, New York, U.S.A..

[Ric03] Richard, J.P. (2003) *Time-delay systems: an overview of some recent advances and open problems*, Automatica, **39**, 1667-1694.

[SO04] Sipahi, R. and Olgac, N. (2004) *A novel stability study on multiple time delayed systems using the root clustering paradigm*, preprint.