DDE-BIFTOOL v. 3.1.1 Manual — Bifurcation analysis of delay differential equations

J. Sieber, K. Engelborghs, T. Luzyanina, G. Samaey, D. Roose

April 5, 2017

Keywords nonlinear dynamics, delay-differential equations, stability analysis, periodic solutions, collocation methods, numerical bifurcation analysis, state-dependent delay.

1. Citation, license, and obtaining the package

DDE-BIFTOOL was started by Koen Engelborghs as part of his PhD at the Computer Science Department of the K.U.Leuven under supervision of Prof. Dirk Roose.

Citation Scientific publications for which the package DDE-BIFTOOL has been used shall mention usage of the package DDE-BIFTOOL, and shall cite the following publications to ensure proper attribution and reproducibility:

- K. Engelborghs, T. Luzyanina, and D. Roose. Numerical bifurcation analysis of delay differential equations using DDE-BIFTOOL, ACM Trans. Math. Softw. 28 (1), pp. 1-21, 2002.
- **(this manual)** J. Sieber, K. Engelborghs, T. Luzyanina, G. Samaey, D. Roose . DDE-BIFTOOL v. 3.1.1 Manual — Bifurcation analysis of delay differential equations, `http://arxiv.org/abs/1406.7144`.

The implementation of normal forms for equilibria is based on

- M. M. Bosschaert, B. Wage and Y. Kuznetsov: Description of the extension `ddebiftool_nmfm` `http://ddebiftool.sourceforge.net/nmfm_extension_description.pdf`, 2015.
- B. Wage: Normal form computations for Delay Differential Equations in DDE-BIFTOOL. Master Thesis, Utrecht University (NL), supervised by Y.A. Kuznetsov (`http://dspace.library.uu.nl/handle/1874/296912`, 2014.
- M. M. Bosschaert: Switching from codimension 2 bifurcations of equilibria in delay differential equations. Master Thesis, Utrecht University (NL), supervised by Y.A. Kuznetsov, `http://dspace.library.uu.nl/handle/1874/334792`, 2016.

All versions of this manual from v. 3.0 onward are available at `http://arxiv.org/abs/1406.7144`.

License The following terms cover the use of the software package DDE-BIFTOOL:

Download Upon acceptance of the above terms, one can obtain the package DDE-BIFTOOL (version 3.1.1) from

https://sourceforge.net/projects/ddebiftool.

Versions up to 3.0 and resources on theoretical background continue to be available (under a different license) on

http://twr.cs.kuleuven.be/research/software/delay/ddebiftool.shtml.

2. Version history

2.1. Changes from 3.1 to 3.1.1

- Small bug fix: when changing focus on plot window during `br_contn`, the online plotting no longer follows focus. This keeps the online bifurcation diagram in the same window. An optional named argument `'plotaxis'` has been added to `br_contn` to explicitly set the plot axes.
- Added support for rotations (phase oscillators). Periodicity is enforced only up to multiples of 2π. Demo ../demos/phase_oscillator/html/phase_oscillator.html shows how one can track rotations. Demo was contributed by Azamat Yeldesbay.
- Demos showing detection and computation of Bodganov-Takens bifurcation in

  ../demos/Holling-Tanner/html/HollingTanner_demo.html

  and cusp in

  ../demos/cusp/html/cusp_demo.html.

  Contributed by M. M. Boschaert and Y. Kuznetsov.
- A description of the mathematical formulas behind the normal form computations is now in nmfm_extension_description.pdf, by M.Bossschaert, B. Wage, Y. Kuznetsov.

2.2. Changes from 3.0 to 3.1

Change of License and move to Sourceforge D. Roose has permitted to change the license to a Sourceforge-compliant BSD License. Thus, code and newest releases from version 3.1 onward are now available from https://sourceforge.net/projects/ddebiftool. Older versions will continue to be available from http://twr.cs.kuleuven.be/research/software/delay/ddebiftool.shtml.

New feature: Normal form computation for bifurcations of equilibria The new functionality is only applicable for equations with constant delay. Normal form coefficients can be computed through the extension `ddebiftool_extra_nmfm`. This extension is included in the standard DDE-BIFTOOL archive, but the additional functions are kept in a separate folder. The following bifurcations are currently supported.

- Hopf bifurcation (coefficient L_1 determining criticality),
- generalized Hopf (Bautin) bifurcation (of codimension two, typically encountered along Hopf curves)
- Zero-Hopf interaction (Gavrilov-Guckenheimer bifurcation, of codimension two, typically encountered along Hopf curves)

- Hopf-Hopf interaction (of codimension two, typically encountered along Hopf curves)

The extension comes with a demo `nmfm_demo`. The demos `neuron`, `minimal_demo` and `Mackey-Glass` illustrate the new functionality, too. Background theory is given in [46].

2.3. Changes from 2.03 to 3.0

New features

- **Continuation of local periodic orbit bifurcations** for systems with constant or state-dependent delay is now supported through the extension `ddebiftool_extra_psol`. This extension is included in the standard DDE-BIFTOOL archive, but the additional functions are kept in a separate folder.
- **User-defined functions** specifying the right-hand side and delays (such as `sys_rhs` and `sys_tau`) can have arbitrary names. These user functions (with arbitrary names) get collected in a structure `funcs`, which gets then passed on to the DDE-BIFTOOL routines. This interface is similar to other functions acting on Matlab functions such as `fzero` or `ode45`. It enables users to add extensions such as `ddebiftool_extra_psol` without changing the core routines.
- **State-dependent delays** can now have arbitrary levels of nesting (for example, periodic orbits of $\dot{x}(t) = \mu - x(t - x(t - x(t - x(t))))$ and their bifurcations can be tracked).
- **Vectorization** Continuation of periodic orbits and their bifurcations benefits (moderately) from vectorization of the user-defined functions.
- **Utilities** Recurring tasks (such as branching off at bifurcations, defining initial pieces of branches, or extracting the number of unstable eigenvalues) can now be performed more conveniently with some auxiliary functions provided in a separate folder `ddebiftool_utilities`. See `../demos/neuron/html/demo1_simple.html` for a demonstration.
- **Bugs fixed** Some bugs and problems have been fixed in the implementation of the heuristics applied to choose the stepsize in the computation of eigenvalues of equilibria [45].
- **Continuation of relative equilibiria and relative periodic orbits** and their local bifurcations for systems with constant delay and rotational symmetry (saddle-node bifurcation, Hopf bifurcation, period-doubling, and torus bifurcation) is now supported through the extension `ddebiftool_extra_rotsym`.

Extensions come with demos and separate documentation.

Change of user interface Versions from 3.0 onward have a different the user interface for many DDE-BIFTOOL functions than versions up to 2.03. They add one additional input argument `funcs` (this new argument comes always *first*). Since DDE-BIFTOOL v. 3.0 changed the user interface, scripts written for DDE-BIFTOOL v. 2.0x or earlier will not work with versions later than 3.0. For this reason version 2.03 will continue to be available. Users of both versions should ensure that only one version is in the Matlab path at any time to avoid naming conflicts.

3. Capabilities and related reading and software

DDE-BIFTOOL consists of a set of routines running in Matlab[1] [29] or octave[2], both widely used environments for scientific computing. The aim of the package is to provide a tool for numerical

[1] `http://www.mathworks.com`
[2] `http://www.gnu.org/software/octave`

bifurcation analysis of steady state solutions and periodic solutions of differential equations with constant delays (called DDEs) or state-dependent delays (here called sd-DDEs). It also allows users to compute homoclinic and heteroclinic orbits in DDEs (with constant delays).

Capabilities DDE-BIFTOOL can perform the following computations:

- continuation of steady state solutions (typically in a single parameter);
- approximation of the rightmost, stability-determining roots of the characteristic equation which can further be corrected using a Newton iteration;
- continuation of steady state folds and Hopf bifurcations (typically in two system parameters);
- continuation of periodic orbits using orthogonal collocation with adaptive mesh selection (starting from a previously computed Hopf point or an initial guess of a periodic solution profile);
- approximation of the largest stability-determining Floquet multipliers of periodic orbits;
- branching onto the secondary branch of periodic solutions at a period doubling bifurcation or a branch point;
- continuation of folds, period doublings and torus bifurcations (typically in two system parameters) using the extension `ddebiftool_extra_psol`;
- computation of normal form coefficients for Hopf bifurcations and codimension-two bifurcations along Hopf bifurcation curves (typically in two system parameters) using the extension `ddebiftool_extra_nmfm`;
- continuation of connecting orbits (using the appropriate number of parameters).

All computations can be performed for problems with an arbitrary number of discrete delays. These delays can be either parameters or functions of the state. (The only exception are computations of connecting orbits, which support only problems with delays as parameters at the moment.)

A practical difference to AUTO or MatCont is that the package does not detect bifurcations automatically because the computation of eigenvalues or Floquet multipliers may require more computational effort than the computation of the equilibria or periodic orbits (for example, if the system dimension is small but one delay is large). Instead the evolution of the eigenvalues can be computed along solution branches in a separate step if required. This allows the user to detect and identify bifurcations.

About this manual — related reading This manual documents version 3.1.1. Earlier versions of the manual for earlier versions of DDE-BIFTOOL continue to be available at the web addresses

versions ≥ 3.0	`http://arxiv.org/abs/1406.7144`
versions ≤ 2.03	`http://twr.cs.kuleuven.be/research/software/delay/ddebiftool.shtml`.

For readers who intend to analyse only systems with constant delays, the parts of the manual related to systems with state-dependent delays can be skipped (sections 5.2, 6.3). In the rest of this manual we assume the reader is familiar with the notion of a delay differential equation and with the basic concepts of bifurcation analysis for ordinary differential equations. The theory on delay differential equations and a large number of examples are described in several books. Most notably the early [4, 13, 14, 25, 32] and the more recent [2, 30, 26, 10, 31]. Several excellent books contain introductions to dynamical systems and bifurcation theory of ordinary differential equations, see, e.g., [1, 6, 23, 33, 40].

Turorial demos The tutorial demos `demo1` and `sd_demo`, providing a step-by-step walk-through for the typical working mode with DDE-BIFTOOL are included as separate `html` files, published directly from the comments in the demo code. See `../demos/index.html` for links to all demos, many of which are extensively commented.

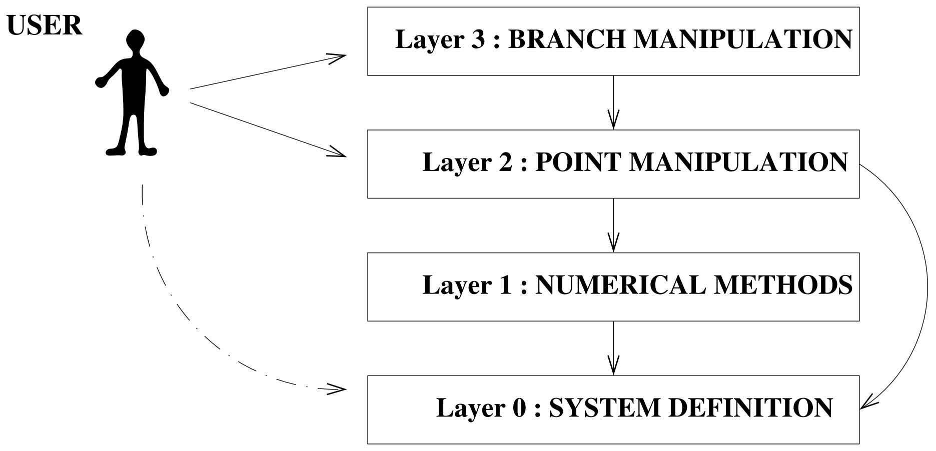
USER
Layer 3 : BRANCH MANIPULATION
Layer 2 : POINT MANIPULATION
Layer 1 : NUMERICAL METHODS
Layer 0 : SYSTEM DEFINITION

Figure 1: The structure of DDE-BIFTOOL. Arrows indicate the calling (−) or writing (·−) of routines in a certain layer.

Related software A large number of packages exist for numerical continuation and bifurcation analysis of systems of ordinary differential equations. Currently maintained packages are

`AUTO`	url: `http://sourceforge.net/projects/auto-07p`	using FORTRAN or C [12, 11],
`MatCont`	url: `http://sourceforge.net/projects/matcont/`	for Matlab [9, 22], and
`Coco`	url: `http://sourceforge.net/projects/cocotools`	for Matlab [8].

For delay differential equations the package

`knut`	url: `http://gitorious.org/knut`	using C++

(formerly `PDDECONT`) is available as a stand-alone package (written in C++, but with a user interface requiring no programming). This package was developed in parallel with DDE-BIFTOOL but independently by R. Szalai [44, 38]. For simulation (time integration) of delay differential equations the reader is, e.g., referred to the packages ARCHI, DKLAG6, XPPAUT, DDVERK, RADAR and dde23, see [37, 7, 21, 20, 41, 24]. Of these, only XPPAUT has a graphical interface (and allows limited stability analysis of steady state solutions of DDEs along the lines of [35]). TRACE-DDE is a Matlab tool (with graphical interface) for linear stability analysis of linear constant-coefficient DDEs [5].

4. Structure of DDE-BIFTOOL

The structure of the package is depicted in figure 1. It consists of four layers.

Layer 0 contains the system definition and consists of routines which allow to evaluate the right hand side f and its derivatives, state-dependent delays and their derivatives and to set or get the parameters and the constant delays. It should be provided by the user and is explained in more detail in section 6. All user-provided functions are collected in a single structure (called `funcs` in this manual), and are passed on by the user as arguments to layer-3 or layer-2 functions. ***Note that this is a change in user interface between version 2.03 and version 3.0!***

Layer 1 forms the numerical core of the package and is (normally) not directly accessed by the user. The numerical methods used are explained briefly in section 10, more details can be found in the papers [35, 18, 17, 16, 19, 34, 39] and in [15]. Its functionality is hidden by and used through layers 2 and 3.

Layer 2 contains routines to manipulate individual points. Names of routines in this layer start with "p_". A point has one of the following five types. It can be a steady state point (abbreviated 'stst'), steady state Hopf (abbreviated 'hopf') or fold (abbreviated 'fold') bifurcation point, a periodic solution point (abbreviated 'psol') or a connecting orbit point (abbreviated 'hcli'). Furthermore, a point can contain additional information concerning its stability. Routines are provided to compute individual points, to compute and plot their stability and to convert points from one type to another.

Layer 3 contains routines to manipulate branches. Names of routines in this layer start with "br_". A branch is structure containing an array of (at least two) points, three sets of method parameters and specifications concerning the free parameters. The 'point' field of a branch contains an array of points of the same type ordered along the branch. The 'method' field contains parameters of the computation of individual points, the continuation strategy and the computation of stability. The 'parameter' field contains specification of the free parameters (which are allowed to vary along the branch), parameter bounds and maximal step sizes. Routines are provided to extend a given branch (that is, to compute extra points using continuation), to (re)compute stability along the branch and to visualize the branch and/or its stability.

Layers 2 and 3 require specific data structures, explained in section 7, to represent points, stability information, branches, to pass method parameters and to specify plotting information. Usage of these layers is demonstrated through a step-by-step analysis of the demo systems neuron, sd_demo and hom_demo (see ../demos/index.html). Descriptions of input/output parameters and functionality of all routines in layers 2 and 3 are given in sections 8 respectively 9.

5. Delay differential equations

This section introduces the mathematical notation that we refer to in this manual to describe the problems solved by DDE-BIFTOOL.

5.1. Equations with constant delays

Consider the system of delay differential equations with constant delays (DDEs),

$$\frac{\mathrm{d}}{\mathrm{d}t}x(t) = f(x(t), x(t-\tau_1), \dots, x(t-\tau_m), \eta), \tag{1}$$

where $x(t) \in \mathbb{R}^n$, $f : \mathbb{R}^{n(m+1)} \times \mathbb{R}^p \to \mathbb{R}^n$ is a nonlinear smooth function depending on a number of parameters $\eta \in \mathbb{R}^p$, and delays $\tau_i > 0$, $i = 1, \dots, m$. Call τ the maximal delay,

$$\tau = \max_{i=1,\dots,m} \tau_i.$$

The linearization of (1) around a solution $x^*(t)$ is the *variational equation*, given by,

$$\frac{\mathrm{d}}{\mathrm{d}t}y(t) = A_0(t)y(t) + \sum_{i=1}^{m} A_i(t)y(t-\tau_i), \tag{2}$$

where, using $f \equiv f(x^0, x^1, \dots, x^m, \eta)$,

$$A_i(t) = \frac{\partial f}{\partial x^i}(x^*(t), x^*(t-\tau_1), \dots, x^*(t-\tau_m), \eta),\ i = 0, \dots, m. \tag{3}$$

5.1.1. Steady states

If $x^*(t)$ corresponds to a steady state solution,

$$x^*(t) \equiv x^* \in \mathbb{R}^n,\ \text{with } f(x^*, x^*, \dots, x^*, \eta) = 0,$$

then the matrices $A_i(t)$ are constant, $A_i(t) \equiv A_i$, and the corresponding variational equation (2) leads to a *characteristic equation*. Define the $n \times n$-dimensional matrix Δ as

$$\Delta(\lambda) = \lambda I - A_0 - \sum_{i=1}^{m} A_i e^{-\lambda \tau_i}. \tag{4}$$

Then the characteristic equation reads,

$$\det(\Delta(\lambda)) = 0. \tag{5}$$

Equation (5) has an infinite number of roots $\lambda \in \mathbb{C}$ which determine the stability of the steady state solution x^*. The steady state solution is (asymptotically) stable provided all roots of the characteristic equation (5) have negative real part; it is unstable if there exists a root with positive real part. It is known that the number of roots in any right half plane $\mathrm{Re}(\lambda) > \gamma$, $\gamma \in \mathbb{R}$ is finite, hence, the stability is always determined by a finite number of roots.

Bifurcations occur whenever roots move through the imaginary axis as one or more parameters are changed. Generically a fold bifurcation (or turning point) occurs when the root is real (that is, equal to zero) and a Hopf bifurcation occurs when a pair of complex conjugate roots crosses the imaginary axis.

5.1.2. Periodic orbits

A periodic solution $x^*(t)$ is a solution which repeats itself after a finite time, that is,

$$x^*(t+T) = x^*(t), \text{ for all } t.$$

Here $T > 0$ is the period. The stability around the periodic solution is determined by the time integration operator $S(T,0)$ which integrates the variational equation (2) around $x^*(t)$ from time $t = 0$ over the period. This operator is called the *monodromy operator* and its (infinite number of) eigenvalues, which are independent of the starting moment $t = 0$, are called the *Floquet multipliers*. Furthermore, if $S(T,0)^k$ is compact for $k > \tau/T$. Thus, there are at most finitely many Floquet multipliers outside of any ball around the origin of the complex plane.

For autonomous systems there is always a *trivial* Floquet multiplier at unity, corresponding to a perturbation along the time derivative of the periodic solution. The periodic solution is exponentially stable provided all multipliers (except the trivial one) have modulus smaller than unity, it is exponenially unstable if there exists a multiplier with modulus larger than unity.

5.1.3. Connecting orbits

We call a solution $x^*(t)$ of (1) at $\eta = \eta^*$ a *connecting orbit* if the limits

$$\lim_{t \to -\infty} x^*(t) = x^-, \qquad \lim_{t \to +\infty} x^*(t) = x^+, \tag{6}$$

exist. For continuous f, x^- and x^+ are steady state solutions. If $x^- = x^+$, the orbit is called homoclinic, otherwise it is heteroclinic.

5.2. Equations with state-dependent delays

Consider the system of delay differential equations with state-dependent delays (sd-DDEs),

$$\begin{cases} \dfrac{\mathrm{d}}{\mathrm{d}t} x(t) = f(x_0, x_1, \ldots, x_m, \eta), \\ \qquad x_j = x(t - \tau_j(x_0, \ldots, x_{j-1}, \eta) \quad (\tau_0 = 0, j = 1, \ldots, m), \end{cases} \tag{7}$$

where $x(t) \in \mathbb{R}^n$, and

$$f : \mathbb{R}^{n(m+1)} \times \mathbb{R}^p \to \mathbb{R}^n$$
$$\tau_j : \mathbb{R}^{nj} \times \mathbb{R}^p \to [0, \infty)$$

are smooth functions depending of their arguments. The right-hand side f depends on $m+1$ states $x_j = x(t-\tau_j) \in \mathbb{R}^n$ $(j = 0, \ldots, m)$ and p parameters $\eta \in \mathbb{R}^p$. The jth delay function τ_j depends on all previously defined $j-1$ states $x(t-\tau_i) \in \mathbb{R}^n$ $(i = 0, \ldots, j-1)$ and p parameters $\eta \in \mathbb{R}^p$. This definition permits the user to formulate sd-DDEs with arbitrary levels of nesting in their function arguments.

The linearization around a solution $(x^*(t), \eta^*)$ of (7) (the *variational equation*) with respect to x is given by (see [27], we are using the notation $x_0^* = x^*(t)$, $\tau_j^*(t) = \tau_j(x_0^*, \ldots, x_{j-1}^*, \eta^*)$ and $x_j^* = x^*(t - \tau_j^*(t))$ for $j \geq 1$)

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t} y(t) &= \sum_{j=0}^{m} A_j(t) Y_k \\ Y_0 &= y(t) \\ Y_j &= y(t - \tau_j^*(t)) - (x^*)'(t - \tau_j(t)) \sum_{k=0}^{j-1} B_{k,j}(t) Y_k, \quad (j = 1 \ldots m)\end{aligned} \tag{8}$$

where $(x^*)'(t) = \mathrm{d}x^*(t)/\mathrm{d}t$, and

$$\begin{aligned}A_j(t) &= \frac{\partial f}{\partial x^j}(x_0^*, x_1^*, \ldots, x_m^*, \eta) \in \mathbb{R}^{n \times n}, \quad (j = 0, \ldots, m), \\ B_{j,k}(t) &= \frac{\partial \tau_j}{\partial x^k}(x_0^*, x_1^*, \ldots, x_{j-1}^*, \eta^*) \in \mathbb{R}^{1 \times n}, \quad (k = 0, \ldots, j-1, j = 1, \ldots, m).\end{aligned} \tag{9}$$

If $(x^*(t), \tilde{\tau}^*(t))$ corresponds to a steady state solution, then $x^*(t) = x_0^* = \ldots = x_m^* \equiv x^* \in \mathbb{R}^n$, and $\tau_j^*(t) \equiv \tau_j(x^*, \ldots, x^*, \eta^*)$ for all $j \geq 1$, with

$$f(x^*, x^*, \ldots, x^*, \eta^*) = 0$$

then the matrices $A_i(t)$ are constant, $A_i(t) \equiv A_i$, and the vectors $B_{i,j}(t)$ consist of zero elements only. In this case, the corresponding variational equation (8) is a constant delay differential equation and it leads to the characteristic equation (5), i.e. a characteristic equation with constant delays. Hence the stability analysis of a steady state solution of (7) is similar to the stability analysis of (1).

Note that the right-hand side f, when considered as a functional mapping a history segment into $\mathbb{R}^n$ is not locally Lipschitz continuous. This creates technical difficulties when considering an sd-DDE of type (7) as an infinite-dimensional system, because the solution does not depend smoothly on the initial condition (see Hartung *et al* [27] for a detailed review). However, periodic boundary-value problems for (7) can be reduced to finite-dimensional systems of algebraic equations that are as smooth as the coefficient functions f and τ_j [43]. This implies that all periodic orbits and their bifurcations and stability as computed by DDE-BIFTOOL behave as expected. In particular, branching off at Hopf bifurcations and period doubling works in the same way as for constant delays (the proof for the Hopf bifurcation in sd-DDEs is also given in [43]).

Moreover, Mallet-Paret and Nussbaum [36] proved that the stability of the linear variational equation (8) indeed reflects the *local stability* of the solution $(x^*(t), \tilde{\tau}^*(t))$ of (7). For details on the relevant theory and numerical bifurcation analysis of differential equations with state-dependent delay see [34, 27] and the references therein.

6. System definition

6.1. Change from DDE-BIFTOOL v. 2.03 to v. 3.0

Note that in DDE-BIFTOOL 3.1.1 all user-provided functions can be arbitrary function handles, collected into a structure using the function `set_funcs`, explained in section 6.5. The only typical mandatory functions for the user to provide are the right-hand side (`'sys_rhs'`) and the function returning the delay indices (`'sys_tau'`). The names of the user functions can be arbitrary, and user functions can be anonymous. This is a change from previous versions. See the tutorials in `../demos/index.html` for examples of usage, and the function description in section 6.5 for details.

6.2. Equations with constant delays

As an illustrative example we will use the following system of delay differential equations, taken from [42],

$$\left\{\begin{array}{l} \dot{x}_1(t) = -\kappa x_1(t) + \beta \tanh(x_1(t-\tau_s)) + a_{12}\tanh(x_2(t-\tau_2)) \\ \dot{x}_2(t) = -\kappa x_2(t) + \beta \tanh(x_2(t-\tau_s)) + a_{21}\tanh(x_1(t-\tau_1)). \end{array}\right. \tag{10}$$

This system models two coupled neurons with time delayed connections. It has two components (x_1 and x_2), three delays (τ_1, τ_2 and τ_s), and four parameters (κ, β, a_{12} and a_{21}). The demo `neuron` (see `../demos/index.html`) walks through the bifurcation analysis of system (10) step by step to demonstrate the working pattern for DDE-BIFTOOL.

To define a system, the user should provide the following Matlab functions, given in the following paragraphs for system (10).

6.2.1. Right-hand side — `sys_rhs`

The right-hand side is a function of two arguments. For our example (10), this would have the form (giving the right-hand side the name `neuron_sys_rhs`)

```
neuron_sys_rhs=@(xx,par)[...
  -par(1)*xx(1,1)+par(2)*tanh(xx(1,4))+par(3)*tanh(xx(2,3));...
  -par(1)*xx(2,1)+par(2)*tanh(xx(2,4))+par(4)*tanh(xx(1,2))];
%par=[\kappa,\beta, a_{12}, a_{21},\tau_1,\tau_2, \tau_s]
```

Listing 1: Definition for right-hand side of (10) as a variable.

Meaning of the arguments of the right-hand side function:

- `xx` $\in \mathbb{R}^{n\times(m+1)}$ contains the state variable(s) at the present and in the past,
- `par` $\in \mathbb{R}^{1\times p}$ contains the parameters, `par` $=\eta$.

The delays τ_i ($i = 1\ldots,m$) are considered to be part of the parameters ($\tau_i = \eta_{j(i)}$, $i = 1,\ldots,m$). This is natural since the stability of steady solutions and the position and stability of periodic solutions depend on the values of the delays. Furthermore delays can occur both as a 'physical' parameter and as delay, as in $\dot{x} = \tau x(t-\tau)$. From these inputs the right hand side f is evaluated at time t. Notice that the parameters have a specific order in par indicated in the comment line.

An alternative (vectorized) form would be

```
  neuron_sys_rhs=@(xx,p)[...
  -p(1)*xx(1,1,:)+p(2)*tanh(xx(1,4,:))+p(3)*tanh(xx(2,3,:));....
  -p(1)*xx(2,1,:)+p(2)*tanh(xx(2,4,:))+p(4)*tanh(xx(1,2,:))];
```

Listing 2: Alternative definition of the right-hand side of (10), vectorized for speed-up of periodic orbit computations.

Note the additional colon in argument `xx` and compare to Listing 1. The form shown in Listing 2 can be called in many points along a mesh simultaneously, speeding up the computations during analysis of periodic orbits.

6.2.2. Delays — `sys_tau`

For constant delays another function is required which returns the *position* of the delays in the parameter list. For our example, this is

```
neuron_tau=@()[5 6 7];
```

This function has no arguments for constant delays, and returns a row vector of indices into the parameter vector.

6.2.3. Jacobians of right-hand side — `sys_deri` (optional, but recommended)

Several derivatives of the right hand side function f need to be evaluated during bifurcation analysis. By default, DDE-BIFTOOL uses a finite-difference approximation, implemented in `df_deriv.m`. For speed-up or in case of convergence difficulties the user may provide the Jacobians of the right-hand side analytically as a separate function. Its header is of the format

```
function J=sys_deri(xx,par,nx,np,v)
```

Arguments:

- `xx` $\in \mathbb{R}^{n\times(m+1)}$ contains the state variable(s) at the present and in the past (as for the right-hand side);
- `par` $\in \mathbb{R}^{1\times p}$ contains the parameters, `par` $= \eta$ (as for the right-hand side);
- `nx` (empty, one integer or two integers) index (indices) of `xx` with respect to which the right-hand side is to be differentiated
- `np` (empty or integer) whether right-hand side is to be differentiated with respect to parameters
- `v` (empty or $\mathbb{C}^n$) for mixed derivatives with respect to `xx`, only the product of the mixed derivative with `v` is needed.

The result `J` is a matrix of partial derivatives of f which depends on the type of derivative requested via `nx` and `np` multiplied with `v` (when nonempty), see table 1.

`J` is defined as follows. Initialize `J` with f. If `nx` is nonempty take the derivative of `J` with respect to those arguments listed in `nx`'s entries. Each entry of `nx` is a number between 0 and m based on $f \equiv f(x^0, x^1, \dots, x^m, \eta)$. E.g., if `nx` has only one element take the derivative with respect to $x^{\texttt{nx(1)}}$. If it has two elements, take, of the result, the derivative with respect to $x^{\texttt{nx(2)}}$ and so on. Similarly, if `np` is nonempty take, of the resulting `J`, the derivative with respect to $\eta_{\texttt{np(i)}}$ where i ranges over all the elements of `np`, $1 \le i \le p$. Finally, if v is not an empty vector multiply the result with v. The latter is used to prevent `J` from being a tensor if two derivatives with respect to state variables are taken (when `nx` contains two elements). Not all possible combinations of these derivatives have to be provided. In the current version, `nx` has at most two elements and `np` at most one. The possibilities are further restricted as listed in table 1.

In the last row of table 1 the elements of `J` are given by,

$$\texttt{J}_{i,j} = \left[\frac{\partial}{\partial x^{\texttt{nx(2)}}} A_{\texttt{nx(1)}} v\right]_{i,j} = \frac{\partial}{\partial x_j^{\texttt{nx(2)}}} \left(\sum_{k=1}^{n} \frac{\partial f_i}{\partial x_k^{\mathsf{nx}(1)}} v_k\right),$$

with A_l as defined in (3).

`length(nx)`	`length(np)`	`v`	`J`
1	0	empty	$\frac{\partial f}{\partial x^{\texttt{nx(1)}}} = A_{\texttt{nx(1)}} \in \mathbb{R}^{n\times n}$
0	1	empty	$\frac{\partial f}{\partial \eta_{\texttt{np(1)}}} \in \mathbb{R}^{n\times 1}$
1	1	empty	$\frac{\partial^2 f}{\partial x^{\texttt{nx(1)}} \partial \eta_{\texttt{np(1)}}} \in \mathbb{R}^{n\times n}$
2	0	$\in \mathbb{C}^{n\times 1}$	$\frac{\partial}{\partial x^{\texttt{nx(2)}}} \left(A_{\texttt{nx(1)}} v\right) \in \mathbb{C}^{n\times n}$

Table 1: Results of the function `sys_deri` depending on its input parameters `nx`, `np` and `v` using $f \equiv f(x^0, x^1, \ldots, x^m, \eta)$.

The resulting routine is quite long, even for the small system (10); see Listing 6 in Appendix A for a printout of the function body. Furthermore, implementing so many derivatives is an activity prone to a number of typing mistakes. Hence a default routine `df_deriv` is available which implements finite difference formulas to approximate the requested derivatives (using several calls to the right-hand side. It is, however, recommended to provide at least the first order derivatives with respect to the state variables using analytical formulas. These derivatives occur in the determining systems for fold and Hopf bifurcations and for connecting orbits, and in the computation of characteristic roots and Floquet multipliers. All other derivatives are only necessary in the Jacobians of the respective Newton procedures and thus influence only the convergence speed.

6.3. Equations with state-dependent delays

DDE-BIFTOOL also permits the delays to depend on parameters and the state. If at least one delay is state-dependent then the format and semantics of the function specifying the delays, `sys_tau`, is different from the format used for constant delays in section 6.2.2 (it now provides the *values* of the delays). Note that for a system with only constant delays we recommend the use of the system definitions as described in section 6.2 to reduce the computational effort.

As an illustrative example we will use the following system of delay differential equations,

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}x_1(t) &= \frac{1}{p_1 + x_2(t)}\left(1 - p_2 x_1(t) x_1(t-\tau_3) x_3(t-\tau_3) + p_3 x_1(t-\tau_1) x_2(t-\tau_2)\right),\\
\frac{\mathrm{d}}{\mathrm{d}t}x_2(t) &= \frac{p_4 x_1(t)}{p_1 + x_2(t)} + p_5 \tanh(x_2(t-\tau_5)) - 1,\\
\frac{\mathrm{d}}{\mathrm{d}t}x_3(t) &= p_6(x_2(t) - x_3(t)) - p_7(x_1(t-\tau_6) - x_2(t-\tau_4))e^{-p_8\tau_5},\\
\frac{\mathrm{d}}{\mathrm{d}t}x_4(t) &= x_1(t-\tau_4)e^{-p_1\tau_5} - 0.1,\\
\frac{\mathrm{d}}{\mathrm{d}t}x_5(t) &= 3(x_1(t-\tau_2) - x_5(t)) - p_9,
\end{aligned} \tag{11}$$

```
function f=sd_rhs(xx,par)
f(1,1)=(1/(par(1)+xx(2,1)))*(1-par(2)*xx(1,1)*xx(1,4)*xx(3,4)+...
        par(3)*xx(1,2)*xx(2,3));
f(2,1)=par(4)*xx(1,1)/(par(1)+xx(2,1))+par(5)*tanh(xx(2,6))-1;
f(3,1)=par(6)*(xx(2,1)-xx(3,1))-par(7)*(xx(1,7)-...
        xx(2,5))*exp(-par(8)*xx(4,1));
f(4,1)=xx(1,5)*exp(-par(1)*xx(4,1))-0.1;
f(5,1)=3*(xx(1,3)-xx(5,1))-par(9);
end
```

Listing 3: Listing of right-hand side `'sys_rhs'` function for (11) function for (11), here called `sd_rhs`.

where

$$\begin{aligned}
&\tau_1, \tau_2 \text{ are constant delays,}\\
&\tau_3 = 2 + p_5\tau_1 x_2(t)x_2(t-\tau_1),\\
&\tau_4 = 1 - \frac{1}{1 + x_1(t)x_2(t-\tau_2)},\\
&\tau_5 = x_4(t),\\
&\tau_6 = x_5(t).
\end{aligned}$$

This system has five components $(x_1, \ldots, x_5)$, six delays $(\tau_1, \ldots, \tau_6)$ and eleven parameters $(p_1, \ldots, p_{11})$, where $p_{10} = \tau_1$ and $p_{11} = \tau_2$. A step-by-step tutorial for analysis of sd-DDEs is given in demo sd_demo (see `../demos/index.html`) of this system using (11).

To define a system with state-dependent delays, the user should provide the following Matlab functions, given in the following sections for system (11).

6.3.1. Right-hand side — `sys_rhs`

The definition and functionality of this routine is equivalent to the one described in section 6.2.1. Notice that the argument `xx` contains the state variable(s) at the present and in the past, `xx` $= [x(t)\ x(t-\tau_1)\ \ldots\ \ldots\ x(t-\tau_m)]$. Possible constant delays (τ_1 and τ_2 in example (11)) are also considered to be part of the parameters. See Listing 3 for the right-hand side to be provided for example (11).

6.3.2. Delays — `sys_tau` and `sys_ntau`

The format and semantics of the routines specifying the delays differ from the one described in section 6.2.2. The user has to provide two functions:

```
function ntau=sys_ntau()
function  tau=sys_tau (ind,xx,par)
```

The function `sys_ntau` has no arguments and returns the number of (constant and state-dependent) delays. For the example (11), this could be the anonymous function `@()6;`.

The function `sys_tau` has the three arguments:

- `ind` (integer ≥ 1) indicates, which delay is to be returned;
- `xx` ($n \times$ `ind`-matrix) is the state: `xx(:,1)`$= x(t)$, `xx(:,k)`$= x(t-\tau_{k-1})$ for $k = 2\ldots$`ind`;
- `par` (row vector) is the vector of system parameters

```
function tau=sd_tau(ind,xx,par)
if ind==1
  tau=par(10);
elseif ind==2
  tau=par(11);
elseif ind==3
  tau=2+par(5)*par(10)*xx(2,1)*xx(2,2);
elseif ind==4
 tau=1-1/(1+xx(2,3)*xx(1,1));
elseif ind==5
 tau=xx(4,1);
elseif ind==6
 tau=xx(5,1);
end
end
```

Listing 4: Listing of `'sys_tau'` function for (11), here called `sd_tau`.

The output is the value of $\tau_{\mathtt{ind}}$, the `ind`'th delay. DDE-BIFTOOL calls `sys_tau` m times where m =`sys_ntau()`. In the first call `xx` is the $n \times 1$ vector, $x(t)$ such that τ_1 may depend on $x(t)$ and `par`. After the first call to `sys_tau`, DDE-BIFTOOL computes $x(t-\tau_1)$. In the second call to `sys_tau` `xx` is a $n \times 2$ matrix, consisting of $[x(t), x(t-\tau_1)]$ such that the delay τ_2 may depend on $x(t)$, $x(t-\tau_1)$ and `par`, etc. In this way, the user can define state-dependent point delays with arbitrary levels of nesting. The delay function for the example (11) is given in Listing 4.

Note: order of delays The order of the delays requested in `sys_tau` corresponds to the order in which they appear in `xx` as passed to the functions `sys_rhs` and `sys_deri`.

Note: difference to section 6.2.2 When calling `sys_tau` for a constant delay, the value of the delay is returned. This is in contrast with the definition of `sys_tau` in section 6.2.2, where the position in the parameter list is returned.

6.3.3. Jacobian of right-hand side — `sys_deri` (optional, but recommended)

The definition and functionality of this routine is equivalent to the one described in section 6.2.3. We do not present here the routine since it is quite long, see the Matlab code `sd_deri.m` in the demo example `sd_demo`. If the user does not provide a function for the Jacobians the finite-difference approximation (`df_deriv.m`) will be used by default. However, as for constant delays, it is recommended to provide at least the first order derivatives with respect to the state variables using analytical formulas.

6.3.4. Jacobians of delays — `sys_dtau` (optional, but recommended)

The routine of the format

```
  function dtau=sd_dtau(ind,xx,par,nx,np)
```

supplies derivatives of all delays with respect to the state and parameters. Its functionality is similar to the function `sys_deri`. Inputs:

- `ind` (integer ≥ 1) the number of the delay,

- `par` ($n \times$ `ind` matrix) is the state: `xx(:,1)`$= x(t)$, `xx(:,k)`$= x(t - \tau_{k-1})$ for $k = 2 \ldots$ `ind`;
- `nx` (empty, one integer or two integers) index (indices) of `xx` with respect to which the right-hand side is to be differentiated;
- `np` (empty or integer) whether right-hand side is to be differentiated with respect to parameters.

The result dtau is a scalar, vector or matrix of partial derivatives of the delay with number `ind`, which depends on the type of derivative requested via `nx` and `np`, see table 2. The resulting routine is quite long, even for the small system (11); see Listing 7 in Appendix A for a printout of the function body.

If the user does not provide a `'sys_dtau'` function then the default routine `df_derit` will be used, which implements finite difference formulas to approximate the requested derivatives (using several calls to `sys_tau`), analogously to `df_deriv`. As in the case of `sys_deri`, it is recommended to provide at least the first order derivatives with respect to the state variables using analytical formulas.

`length(nx)`	`length(np)`	`dtau`
1	0	$\frac{\partial \tau_{\mathtt{ind}}}{\partial x^{\mathtt{nx(1)}}} \in \mathbb{R}^n$
0	1	$\frac{\partial \tau_{\mathtt{ind}}}{\partial\ \eta_{\mathtt{np(1)}}} \in \mathbb{R}$
1	1	$\frac{\partial^2 \tau_{\mathtt{ind}}}{\partial x^{\mathtt{nx(1)}}\ \partial \eta_{\mathtt{np(1)}}} \in \mathbb{R}^n$
2	0	$\frac{\partial}{\partial x^{\mathtt{nx(2)}}} \left[\frac{\partial \tau_{\mathtt{ind}}}{\partial x^{\mathtt{nx(1)}}} \right] \in \mathbb{R}^{n \times n}$

Table 2: Results of the function `sys_dtau` depending on its input parameters `nx` and `np` (`ind`$= 1, \ldots, m$).

6.4. Extra conditions — `sys_cond`

A system routine `sys_cond` with a header of the type

```
function [res,p]=sys_cond(point)
```

can be used to add extra conditions during corrections and continuation, see section 10.2 for an explanation of arguments and outputs.

6.5. Collecting user functions into a structure — call `set_funcs`

The user-provided functions are passed on as an additional argument to all routines of DDE-BIFTOOL (similar to standard Matlab routines such as `ode45`). This was changed in DDE-BIFTOOL 3.0 from previous versions. The additional argument is a structure `funcs` containing all the handles to all user-provided functions. In order to create this structure the user is recommended to call the function `set_funcs` at the beginning of the script performing the bifurcation analysis:

```
function funcs=set_funcs(...)
```

Its argument format is in the form of name-value pairs (in arbitrary order, similar to options at the end of a call to `plot`). For the example (10) of a neuron, discussed in section 6.2 and in demo `neuron` (see `../demos/index.html`), the call to `set_funcs` could look as follows:

```
funcs=set_funcs('sys_rhs',neuron_sys_rhs,'sys_tau',@()[5,6,7],...
                'sys_deri',@neuron_sys_deri);
```

Note that `neuron_sys_rhs` is a variable (a function handle pointing to an anonymous function defined as in section 6.2.1), and `neuron_sys_deri.m` is the filename in which the function providing the system derivatives are defined (see section 6.2.3). The delay function `'sys_tau'` is directly specified as an anonymous function in the call to `set_funcs` (not needing to be defined in a separate file or as a separate variable). If one does wish to not provide analytical derivatives, one may drop the `'sys_deri'` pair (then a finite-difference approximation, implemented in `df_deriv`, is used):

```
funcs=set_funcs('sys_rhs',neuron_sys_rhs,'sys_tau',@()[5,6,7]);
```

For the sd-DDE example (11), the call could look as follows:

```
funcs=set_funcs('sys_rhs',@sd_rhs,'sys_tau',@sd_tau,...
    'sys_ntau',@()6,'sys_deri',@sd_deri,'sys_dtau',@sd_dtau);
```

Possible names to be used in the argument sequence equal the resulting field names in `funcs` (see Table 3 in section 7.1 later):

- `'sys_rhs'`: handle of the user function providing the right-hand side, described in sections 6.2.1 and 6.3.1;
- `'sys_tau'`: handle of the user function providing the indices of the delays in the parameter vector for DDEs with constant delays, described in sections 6.2.2, or providing the values of the delays for sd-DDEs as described in section 6.3.2;
- `'sys_ntau'` (relevant for sd-DDEs only): handle of the user function providing the number of delays in sd-DDEs as described in section 6.3.2;
- `'sys_deri'`: handle of the user function providing the Jacobians of right-hand side, described in sections 6.2.3 and 6.3.3;
- `'sys_dtau'` (relevant for sd-DDEs only): handle of the user function providing the Jacobians of delays (for sd-DDEs), described in section 6.2.3;
- `'x_vectorized'` (logical, default `false`): if the functions in `'sys_rhs'`, `'sys_deri'` (if provided), `'sys_tau'` (for sd-DDEs) and `'sys_dtau'` (for sd-DDEs if provided) can be called with 3d arrays in their `xx` argument. Vectorization will speed up computations for periodic orbits only.

An example for a necessary modification of the right-hand side to permit vectorization is given for the neuron example in Listing 2. The output `funcs` is a structure containing all user-provided functions and defaults for the Jacobians if they are not provided. This output is passed on as first argument to all DDE-BIFTOOL routines during bifurcation analysis.

7. Data structures

In this section we describe the data structures used to define the problem, and to present individual points, stability information, branches of points, method parameters and plotting information.

7.1. Problem definition (functions) structure

The user-provided functions described in Section 6 get passed on to DDE-BIFTOOL's routines collected in a single argument `funcs`, a structure containing at least the fields listed in Table 3. Only fields marked with **[!]** are mandatory (for sd-DDEs `'sys_ntau'` is mandatory, too). The user does

field	content	default
`'sys_rhs'` **(mandatory)**	function handle	function in file `sys_rhs.m` if found in current working folder
`'sys_ntau'`	function handle	`@()0`
`'sys_tau'` **(mandatory)**	function handle	function in file `sys_tau.m` if found in current working folder
`'sys_cond'`	function handle	`@(p)dummy_cond`, a built-in routine that adds no conditions
`'sys_deri'`	function handle	`df_deriv`, coming with DDE-BIFTOOL and using finite-difference approximation
`'sys_dtau'`	function handle	`df_derit`, coming with DDE-BIFTOOL and using finite-difference approximation
`'x_vectorized'`	logical	`false`
(`'tp_del'`)	logical	n/a (automatically determined from the number of arguments expected by `funcs.sys_tau`)
(`'sys_deri_provided'`)	logical	n/a (set automatically)
(`'sys_dtau_provided'`)	logical	n/a (set automatically)

Table 3: **Problem definition structure** containing (at least) the user-provided functions. Fields in brackets should not normally be set or manually changed by the user. See also section 6.5.

usually not have to set the fields of this structure manually, but calls the routine `set_funcs`, which returns the structure `funcs` to be passed on to other functions. The usage of `set_funcs` and the meaning of the fields of its output are described in detail in section 6.5. See also the tutorial demos `neuron` and `sd_demo` (see `../demos/index.html` for examples of usage.

7.2. Point structures

Table 4 describes the structures used to represent a single steady state, fold, Hopf, periodic and homoclinic/heteroclinic solution point.

A steady state solution is represented by the parameter values η (which contain also the constant delay values, see section 6) and x^*. A fold bifurcation is represented by the parameter values η, its position x^* and a null-vector of the characteristic matrix $\Delta(0)$. A Hopf bifurcation is represented by the parameter values η, its position x^*, a frequency ω and a (complex) null-vector of the characteristic matrix $\Delta(\mathrm{i}\omega)$.

A periodic solution is represented by the parameter values η, the period T and a time-scaled profile $x^*(t/T)$ on a mesh in [0,1]. The mesh is an ordered collection of *interval points* $\{0 = t_0 < t_1 < \ldots < t_L = 1\}$ and *representation points* $t_{i+\frac{j}{d}}, i = 0, \ldots, L-1, j = 1, \ldots, d-1$ which need to be chosen in function of the interval points as

$$t_{i+\frac{j}{d}} = t_i + \frac{j}{d}(t_{i+1} - t_i).$$

Warning: this assumption is not checked but needs to be fulfilled for correct results! The profile is a continuous piecewise polynomial on the mesh. More specifically, it is a polynomial of degree d on each subinterval $[t_i, t_{i+1}]$, $i = 0, \ldots, L-1$. Each of these polynomials is uniquely represented by its

field	content
`'kind'`	`'stst'`
`'parameter'`	$\mathbb{R}^{1\times p}$
`'x'`	$\mathbb{R}^{n\times 1}$
`'stability'`	empty or struct

(a) Steady state

field	content
`'kind'`	`'fold'`
`'parameter'`	$\mathbb{R}^{1\times p}$
`'x'`	$\mathbb{R}^{n\times 1}$
`'v'`	$\mathbb{R}^{n\times 1}$
`'stability'`	empty or struct

(b) Steady state fold

field	content
`'kind'`	`'hopf'`
`'parameter'`	$\mathbb{R}^{1\times p}$
`'x'`	$\mathbb{R}^{n\times 1}$
`'v'`	$\mathbb{C}^{n\times 1}$
`'omega'`	$\mathbb{R}$
`'stability'`	empty or struct

(c) Steady state Hopf

field	content
`'kind'`	`'psol'`
`'parameter'`	$\mathbb{R}^{1\times p}$
`'mesh'`	$[0,1]^{1\times(Ld+1)}$ or empty
`'degree'`	$\mathbb{N}_0$
`'profile'`	$\mathbb{R}^{n\times(Ld+1)}$
`'period'`	$\mathbb{R}_0^+$
`'stability'`	empty or struct

(d) Periodic orbit

field	content
`'kind'`	`'hcli'`
`'parameter'`	$\mathbb{R}^{1\times p}$
`'mesh'`	$[0,1]^{1\times(Ld+1)}$ or empty
`'degree'`	$\mathbb{N}_0$
`'profile'`	$\mathbb{R}^{n\times(Ld+1)}$
`'period'`	$\mathbb{R}_0^+$
`'x1'`	$\mathbb{R}^n$
`'x2'`	$\mathbb{R}^n$
`'lambda_v'`	$\mathbb{C}^{s_1}$
`'lambda_w'`	$\mathbb{C}^{s_2}$
`'v'`	$\mathbb{C}^{n\times s_1}$
`'w'`	$\mathbb{C}^{n\times s_2}$
`'alpha'`	$\mathbb{C}^{s_1}$
`'epsilon'`	$\mathbb{R}$

(e) Connecting orbit

Table 4: **Point structures**: Field names and corresponding content for the point structures used to represent steady state solutions, fold and Hopf points, periodic solutions and connecting orbits. Here, n is the system dimension, p is the number of parameters, L is the number of intervals used to represent the periodic solution, d is the degree of the polynomial on each interval, s_1 is the number of unstable modes of x^- and s_2 is the number of unstable modes of x^+.

values at the points $\{t_{i+\frac{j}{d}}\}_{j=0,\ldots,d}$. Hence the complete profile is represented by its value at all the mesh points,

$$x^*(t_{i+\frac{j}{d}}),\ i=0,\ldots,L-1,\ j=0,\ldots,d-1;\ \text{and}\ x^*(t_L).$$

Because polynomials on adjacent intervals share the value at the common interval point, this representation is automatically continuous (it is, however, not continuously differentiable). (As indicated in table 4, the mesh may be empty, which indicates the use of an equidistant, fixed mesh.)

A connecting orbit is represented by the parameter values η, the period T, a time-scaled profile $x^*(t/T)$ on a mesh in [0,1], the steady states x^- and x^+ (fields `'x1'` and `'x2'` in the data structure), the unstable eigenvalues of these steady states, λ^- and λ^+ (fields `'lambda_v'` and `'lambda_w'` in the data structure), the unstable right eigenvectors of x^- (`'v'`), the unstable left eigenvectors of x^+ (`'w'`), the direction in which the profile leaves the unstable manifold, determined by α, and the distance of the first point of the profile to x^-, determined by ϵ. For the mesh and profile, the same remarks as in the case of periodic solutions hold.

The point structures are used as input to the point manipulation routines (layer 2) and are used inside the branch structure (see further). The order of the fields in the point structures is important (because they are used as elements of an array inside the branch structure). No such restriction holds for the other structures (method, plot and branch) described in the rest of this section.

7.3. Stability structures

Most of the point structures contain a field `'stability'` storing eigenvalues or Floquet multipliers. (The exception is the `'hcli'` structure for which stability does not really make sense.) During bifurcation analysis the computation of stability is typically performed as a separate step, after computation of the solution branches, because stability computation can easily be more expensive than the solution finding. If no stability has been computed yet, the field `'stability'` is empty, otherwise, it contains the computed stability information in the form described in Table 5.

field	content
`'h'`	$\mathbb{R}$
`'l0'`	$\mathbb{C}^{n_l}$
`'l1'`	$\mathbb{C}^{n_c}$
`'n1'`	$(\{-1\} \cup \mathbb{N}_0)^{n_c}$ or empty

(a) Structure in field `'stability'` for steady state, fold and Hopf points of Table 4

field	content
`'mu'`	$\mathbb{C}^{n_m}$

(b) Structure in field `'stability'` for periodic orbit points of Table 4

Table 5: **Stability structures** for roots of the characteristic equation (in steady state, fold and Hopf structures) (left) and for Floquet multipliers (in the periodic solutions structure) (right). Here, n_l is the number of approximated roots, n_c is the number of corrected roots and n_m is the number of Floquet multipliers.

For steady state, fold and Hopf points, approximations to the rightmost roots of the characteristic equation are provided in field `'l0'` in order of decreasing real part. The steplength that was used to obtain the approximations is provided in field `'h'`. Corrected roots are provided in field `'l1'` and the number of Newton iterations applied for each corrected root in a corresponding field `'n1'`. If unconverged roots are discarded, `'n1'` is empty and the roots in `'l1'` are ordered with respect to real part; otherwise the order in `'l1'` corresponds to the order in `'l0'` and an element -1 in `'n1'` signals that no convergence was reached for the corresponding root in `'l0'` and the last computed iterate is stored in `'l1'`. The collection of uncorrected roots presents more accurate yet less robust information than the collection of approximate roots, see section 10. For periodic solutions only approximations to the Floquet multipliers are provided in a field `'mu'` (in order of decreasing modulus). As the characteristic matrix is not analytically available, DDE-BIFTOOL does not offer an additional correction.

7.4. Method parameters

To compute a single steady state, fold, Hopf, periodic or connecting orbit solution point, several method parameters have to be passed to the appropriate routines. These parameters are collected into a structure with the fields given in Table 6.

For the computation of periodic solutions, additional fields are necessary, marked with and asterisk (*) in Table 6. The meaning of the different fields in Table 6 is explained in section 10.

Parameters controlling the pseudo-arclength continuation (using secant approximations for tangents) are stored in a structure of the form given in Table 8. Similarly, for the approximation and correction of roots of the characteristic equation respectively for the computation of the Floquet multipliers method parameters are passed using a structure of the form given in table 7.

field	content	default value
'newton_max_iterations'	$\mathbb{N}_0$	5, 5, 5, 5, 10
'newton_nmon_iterations'	$\mathbb{N}$	1
'halting_accuracy'	$\mathbb{R}^+$	1e-10, 1e-9, 1e-9, 1e-8, 1e-8
'minimal_accuracy'	$\mathbb{R}_0^+$	1e-8, 1e-7, 1e-7, 1e-6, 1e-6
'extra_condition'	$\{0,1\}$	0
'print_residual_info'	$\{0,1\}$	0
*'phase_condition'	$\{0,1\}$	1
*'collocation_parameters'	$[0,1]^d$ or empty	empty
*'adapt_mesh_before_correct'	$\mathbb{N}$	0
*'adapt_mesh_after_correct'	$\mathbb{N}$	3

Table 6: **Point method structure**: fields and possible values. When different, default values are given in the order 'stst','fold','hopf','psol', 'hcli'. Fields marked with and asterisk (*) are needed and present for points of type 'psol' and 'hcli' only.

7.5. Branch structures

A branch consists of an ordered array of points (all of the same type), and three method structures containing point method parameters, continuation parameters respectively stability computation

field	content	default value
For steady state, fold and Hopf		
'lms_parameter_alpha'	$\mathbb{R}^k$	time_lms('bdf',4)
'lms_parameter_beta'	$\mathbb{R}^k$	time_lms('bdf',4)
'lms_parameter_rho'	$\mathbb{R}_0^+$	time_saf(alpha,beta,0.01,0.01)
'interpolation_order'	$\mathbb{N}_0$	4
'minimal_time_step'	$\mathbb{R}_0^+$	0.01
'maximal_time_step'	$\mathbb{R}_0^+$	0.1
'max_number_of_eigenvalues'	$\mathbb{N}_0$	100
'minimal_real_part'	$\mathbb{R}$ or empty	empty
'max_newton_iterations'	$\mathbb{N}$	6
'root_accuracy'	$\mathbb{R}_0^+$	1e-6
'remove_unconverged_roots'	$\{0,1\}$	1
'delay_accuracy'	$\mathbb{R}_0^-$	-1e-8
For periodic orbit		
'collocation_parameters'	$[0,1]^d$ or empty	empty
'max_number_of_eigenvalues'	$\mathbb{N}$	100
'minimal_modulus'	$\mathbb{R}^+$	0.01
'delay_accuracy'	$\mathbb{R}_0^-$	-1e-8

Table 7: **Stability method structures**: fields and possible values for the approximation and correction of roots of the characteristic equation (top), or for the approximation Floquet multipliers (bottom). The LMS-parameters are default set to the fourth order backwards differentiation LMS-method. The last row in both parts is only used for sd-DDEs.

field	content	default value
'steplength_condition'	$\{0,1\}$	1
'plot'	$\{0,1\}$	1
'prediction'	$\{1\}$	1
'steplength_growth_factor'	$\mathbb{R}_0^+$	1.2
'plot_progress'	$\{0,1\}$	1
'plot_measure'	struct or empty	empty
'halt_before_reject'	$\{0,1\}$	0

Table 8: **Continuation method structure**: fields and possible values.

field	subfield	content	
'point'		array of points	(s. Table 4)
'method'	'point'	point method struct	(s. Table 6)
	'stability'	stability method struct	(s. Table 7)
	'continuation'	continuation method struct	(s. Table 8)
'parameter'	'free'	$\mathbb{N}^{p_f}$	
	'min_bound'	$[\mathbb{N}\ \mathbb{R}]^{p_i}$	
	'max_bound'	$[\mathbb{N}\ \mathbb{R}]^{p_a}$	
	'max_step'	$[\mathbb{N}\ \mathbb{R}]^{p_s}$	

Table 9: **Branch structure**: fields and possible values. Here, p_f is the number of free parameters; p_i, p_a and p_s are the number of minimal parameter values, maximal parameter values respectively maximal parameter steplength values. If any of these values are zero, the corresponding subfield is empty.

parameters, see table 9.

The branch structure has three fields. One, called 'point', which contains an array of point structures, one, called 'method', which is itself a structure containing three subfields and a third, called 'parameter' which contains four subfields. The three subfields of the method field are again structures. The first, called 'point', contains point method parameters as described in Table 6. The second, called 'stability', contains stability method parameters as described in Table 7 and the third, called 'continuation', contains continuation method parameters as described in Table 8. Hence the branch structure incorporates all necessary method parameters which are thus automatically kept when saving a branch variable to file. The parameter field contains a list of free parameter numbers which are allowed to vary during computations, and a list of parameter bounds and maximal steplengths. Each row of the bound and steplength subfields consists of a parameter number (first element) and the value for the bound or steplength limitation. Examples are given in demo `neuron` (see `../demos/index.html`).

A default, empty branch structure can be obtained by passing a list of free parameters and the point kind (as 'stst', 'fold', 'hopf', 'psol' or 'hcli') to the function `df_brnch`. A minimal bound zero is then set for each constant delay if the function `sys_tau` is defined as in section 6.2 (i.e. for DDEs). The method contains default parameters (containing appropriate point, stability and continuation fields) obtained from the function `df_mthod` with as only argument the type of solution point.

field	content	meaning
'field'	{'parameter','x','v','omega',... 'profile','period','stability' ...}	first field to select from a point struct
'subfield'	{'','l0','l1','mu'}	empty string or 2nd field to select
'row'	$\mathbb{N}$ or {'min','max','mean','ampl','all'}	row index
'col'	$\mathbb{N}$ or {'min','max','mean','ampl','all'}	column index
'func'	{'','real','imag','abs'}	function to apply

Table 10: **Measure structure**: fields, content and meaning of a structure describing a measure of a point.

7.6. Scalar measure structure

After a branch has been computed some possibilities are offered to plot its content. For this a (scalar) measure structure is used which defines what information should be taken and how it should be processed to obtain a measure of a given point (such as the amplitude of the profile of a periodic solution, etc...); see Table 10. The result applied to a variable `point` is to be interpreted as

```
scalar_measure=func(point.field.subfield(row,col));
```

where `'field'` presents the field to select, `'subfield'` is empty or presents the subfield to select, 'row' presents the row number or contains one of the functions mentioned in table 10. These functions are applied columnwise over all rows. The function `'all'` specifies that the all rows should be returned. The meaning of `'col'` is similar to `'row'` but for columns. To avoid ambiguity it is required that either `'row'` or `'col'` contains a number or that both contain the function `'all'`. If nonempty, the function 'func' is applied to the result. Note that `'func'` can be a standard Matlab function as well as a user written function. Note also that, when using the value `'all'` in the fields `'col'` and/or `'row'` it is possible to return a non-scalar measure (possibly but not necessarily further processed by `'func'`).

8. Point manipulation

Several of the point manipulation routines have already been used in the previous section. Here we outline their functionality and input and output parameters. A brief description of parameters is also contained within the source code and can be obtained in Matlab using the help command. Note that a vector of zero elements corresponds to an empty matrix (written in Matlab as []).

```
function [point,success]=p_correc(...
           funcs,point0,free_par,step_cnd,method,adapt,previous,d_nr,tz)
```

Function `p_correc` corrects a given point.

- `funcs`: structure of user-defined functions, defining the problem (created, for example, using `set_funcs`).
- `point0`: initial, approximate solution point as a point structure (see table 4).
- `free_par`: a vector of zero, one or more free parameters.
- `step_cnd`: a vector of zero, one or more linear steplength conditions. Each steplength condition is assumed fulfilled for the initial point and hence only the coefficients of the condition with respect to all unknowns are needed. These coefficients are passed as a point structure (see table 4). This means that for, e.g., a steady state solution point `p` the *i*-th steplength condition enforces that

```
step_cnd(i).parameter*(p.parameter-point0.parameter)'+...
step_cnd(i).x'*(p.x-point0.x)
```

is zero. Similar formulas hold for the other solution types.

- `method`: a point method structure containing the method parameters (see table 6).
- `adapt` (optional): if zero or absent, do not use adaptive mesh selection (for periodic solutions); if one, correct, use adaptive mesh selection and recorrect.
- `previous` (optional): for periodic solutions and connecting orbits: if present and not empty, minimize phase shift with respect to this point. Note that this argument should always be present when correcting solutions for sd-DDEs, since in that case the argument `d_nr` always needs to be specified. In the case of steady state, fold or Hopf-like points, one can just enter an empty vector.
- `d_nr`: (only for equations with state-dependent delays) if present, number of a negative state-dependent delay.
- `tz`: (only for equations with state-dependent delays and periodic solutions) if present, a periodic solution is computed such that $\tau_{\mathtt{tz}} = 0$ and $\mathrm{d}\tau_{\mathtt{tz}}/\mathrm{d}t = 0$, where τ is a negative state-dependent delay with number `d_nr`. For steady state solutions, a solution corresponding to $\tau = 0$ is computed.
- `point`: the result of correcting `point0` using the method parameters, steplength condition(s) and free parameter(s) given. Stability information present in `point0` is not passed onto `point`. If divergence occurred, `point` contains the final iterate.
- `success`: nonzero if convergence was detected (that is, if the requested accuracy has been reached).

```
function stability=p_stabil(funcs,point,method)
```

Function `p_stabil` computes stability of a given point by approximating its stability-determining eigenvalues.

- `funcs`: structure of user-defined functions, defining the problem (created, for example, using `set_funcs`).
- `point`: a solution point as a point structure (see table 4).
- `method`: a stability method structure (see table 7).
- `stability`: the computed stability of the point through a collection of approximated eigenvalues (as a structure described in table 5). For steady state, fold and Hopf points both approximations and corrections to the rightmost roots of the characteristic equation are provided. For periodic solutions approximations to the dominant Floquet multipliers are computed.

```
function p_splot(point)
```

Function `p_splot` plots the characteristic roots respectively Floquet multipliers of a given point (which should contain nonempty stability information). Characteristic root approximations and Floquet multipliers are plotted using '$\times$', corrected characteristic roots using '$*$'.

```
function stst_point=p_tostst(funcs,point)
function fold_point=p_tofold(funcs,point)
function hopf_point=p_tohopf(funcs,point,excludefreqs)
function [psol_point,stepcond]=p_topsol(funcs,point,ampl,degree,nr_int)
function [psol_point,stepcond]=p_topsol(funcs,point,ampl,coll_points)
function [psol_point,stepcond]=p_topsol(funcs,hcli_point)
function hcli_point=p_tohcli(funcs,point)
```

The functions `p_tostst`, `p_tofold`, `p_tohopf`, `p_topsol` and `p_tohcli` convert a given point into an approximation of a new point of the kind indicated by their name. They are used to switch from a steady state point to a Hopf point or fold point, from a Hopf point to a fold point or vice versa, from a (nearby double) Hopf point to the second Hopf point, from a Hopf point to the emanating branch of periodic solutions, from a periodic solution near a period doubling bifurcation to the period-doubled branch and from a periodic solution near a homoclinic orbit to this homoclinic orbit. The function `p_tostst` is also capable of extracting the initial and final steady states from a connecting orbit.

The additional argument `excludefreqs` of function `p_tohopf` controls which Hopf frequency is chosen if several are possible. Function `p_tohopf` calls `p_correc` with an initial guess for the Hopf freqency equal to the eigenvalue closest to the imaginary axis. For each entry in the vector `excludefreqs` (of non-negative real numbers) the eigenvalue with imaginary part closest to this entry will be removed from consideration. This becomes useful in systems with large delays. Then equilibria have a large number of eigenvalues close to the imaginary axis, and correspondingly, the system experiences many Hopf bifurcations over short parameter intervals. These Hopf bifurcations can be picked up by calling `p_tohopf` repeatedly, and excluding frequencies one after another.

When starting a periodic solution branch from a Hopf point, an equidistant mesh is produced with `nr_int` intervals and piecewise polynomials of degree `col_degree` and a steplength condition `stepcond` is returned which should be used (together with a corresponding free parameter) in correcting the returned point. This steplength condition (normally) prevents convergence back to the steady state solution (as a degenerate periodic solution of amplitude zero). When jumping to a period-doubled branch, a period-doubled solution profile is produced using `coll_points` for collocation points and a mesh which is the (scaled) concatenation of two times the original mesh. A steplength condition is returned which (normally) prevents convergence back to the single period branch.

When jumping from a homoclinic orbit to a periodic solution, the steplength condition prevents divergence, by keeping the period fixed. When extracting the steady states from a connecting orbit, an array is returned in which the first element is the initial steady state, and the second element is the final steady state.

```
function rm_point=p_remesh(point,new_degree,new_mesh)
```

Function `p_remesh` changes the piecewise polynomial representation of a given periodic solution point.

- `point`: initial point, containing old mesh, old degree and old profile.
- `new_degree`: new degree of piecewise polynomials.
- `new_mesh`: mesh for new representation of periodic solution profile either as a (non-scalar) row vector of mesh points (both interval and representation points, with the latter chosen equidistant between the former, see section 7) or as the new number of intervals. In the latter case the new mesh is adaptively chosen based on the old profile.
- `rm_point`: returned point containing new degree, new mesh and an appropriately interpolated (but uncorrected!) profile.

```
function tau_eva=p_tau(funcs,point,d_nr,t)
```

Function `p_tau` evaluates state-dependent delay(s) with number(s) `d_nr`.

- `funcs`: structure of user-defined functions, defining the problem (created, for example, using `set_funcs`).
- `point`: a solution point as a point structure.
- `d_nr`: number(s) of delay(s) (in increasing order) to evaluate.
- `t` (absent for steady state solutions and optional for periodic solutions): mesh (a time point or a number of time points). If present, delay function(s) are evaluated at the points of `t`, otherwise at the `point.mesh` (if `point.mesh` is empty, an equidistant mesh is used).
- `tau_eva`: evaluated values of delays (at `t`).

The following routines are used within branch routines but are less interesting for the general user.

```
function sc_measure=p_measur(p,measure)
```

Function `p_measur` computes the (scalar) measure `measure` of the given point `p` (see table 10).

```
function p=p_axpy(a,x,y)
```

Function `p_axpy` performs the axpy-operation on points. That is, it computes `p=ax+y` where `a` is a scalar, and `x` and `y` are two point structures of the same type. `p` is the result of the operation on all appropriate fields of the given points. If `x` and `y` are solutions on different meshes, interpolation is used and the result is obtained on the mesh of `x`. Stability information, if present, is not passed onto `p`.

```
function n=p_norm(point)
```

Function `p_norm` computes some norm of a given point structure.

```
function normalized_p=p_normlz(p)
```

Function `p_normlz` performs some normalization on the given point structure `p`. In particular, fold, Hopf and connecting orbit determining eigenvectors are scaled to norm 1.

```
function [delay_nr,tz]=p_tsgn(point)
```

Function `p_tsgn` detects a first negative state-dependent delay.

- `point`: a solution point as a point structure.
- `delay_nr`: number of the first (and only the first !) detected negative delay τ.
- `tz` (only for periodic solutions): $\texttt{tz} \in [0,1]$ is a (time) point such that the delay function $\tau(t)$ has its minimal value near this point. To compute `tz`, a refined mesh is used in the neighbourhood of the minimum of the delay function. This point is later used to compute a periodic solution such that $\tau_{\texttt{tz}} = 0$ and $\mathrm{d}\tau_{\texttt{tz}}/\mathrm{d}t = 0$.

9. Branch manipulation

Usage of most of the branch manipulation routines is illustrated in the demos `neuron` and `sd_demo` (see `../demos/index.html`). Here we outline their functionality and input and output variables. As for all routines in the package, a brief description of the parameters is also contained within the source code and can be obtained in Matlab using the `help` command.

```
function [c_branch,succ,fail,rjct]=br_contn(funcs,branch,max_tries)
```

The function `br_contn` computes (or rather extends) a branch of solution points.

- `funcs`: structure of user-defined functions, defining the problem (created, for example, using `set_funcs`).
- `branch`: initial branch containing at least two points and computation, stability and continuation method parameter structures and a free parameter structure as described in table 9.
- `max_tries`: maximum number of corrections allowed.
- `c_branch`: the branch returned contains a copy of the initial branch plus the extra points computed (starting from the end of the point array in the initial branch).
- `succ`: number of successful corrections.
- `fail`: number of failed corrections.
- `rjct`: number of rejected points.

Note also that successfully computed points are normalized using the procedure `p_normlz` (see section 8).

```
function br_plot(branch,x_measure,y_measure,line_type)
```

Function `br_plot` plots a branch (in the current figure).

- `branch`: branch to plot (see table 9).
- `x_measure`: (scalar) measure to produce plotting quantities for the x-axis (see table 10). If empty, the point number is used to plot against.
- `y_measure`: (scalar) measure to produce plotting quantities for the y-axis (see table 10). If empty, the point number is used to plot against.
- `line_type` (optional): line type to plot with.

```
function [x_measure,y_measure]=df_measr(stability,branch)
function [x_measure,y_measure]=df_measr(stability,par_list,kind)
```

Function `df_measr` returns default measures for plotting.

- `stability`: nonzero if measures are required to plot stability information.
- `branch`: a given branch (see table 9) for which default measures should be constructed.
- `par_list`: a list of parameters for which default measures should be constructed.
- `kind`: a point type for which default measures should be constructed.
- `x_measure`: default scalar measure to use for the x-axis. `x_measure` is chosen as the first parameter which varies along the branch or as the first parameter of `par_list`.
- `y_measure`: default scalar measure to use for the y-axis. If `stability` is zero, the following choices are made for `y_measure`. For steady state solutions, the first component which varies along the branch; for fold and Hopf bifurcations the first parameter value (different from the one used for `x_measure`) which varies along the branch. For periodic solutions, the amplitude of the fist varying component. If `stability` is nonzero, `y_measure` selects the real part of the characteristic roots (for steady state solutions, fold and Hopf bifurcations) or the modulus of the Floquet multipliers (for periodic solutions).

```
function st_branch=br_stabl(funcs,branch,skip,recompute)
```

Function `br_stabl` computes stability information along a previously computed branch.

- `funcs`: structure of user-defined functions, defining the problem (created, for example, using `set_funcs`).
- `branch`: given branch (see table 9).
- `skip`: number of points to skip between stability computations. That is, computations are performed and stability field is filled in every `skip`+1-th point.
- `recompute`: if zero, do not recompute stability information present. If nonzero, discard and recompute old stability information present (for points which were not skipped).
- `st_branch`: a copy of the given branch whose (non-skipped) points contain a non-empty stability field with computed stability information (using the method parameters contained in `branch`).

```
function t_branch=br_rvers(branch)
```

To continue a branch in the other direction (from the beginning instead of from the end of its point array), `br_rvers` reverses the order of the points in the branches point array.

```
function recmp_branch=br_recmp(funcs,branch,point_numbers)
```

Function `br_recmp` recomputes part of a branch.

- `funcs`: structure of user-defined functions, defining the problem (created, for example, using `set_funcs`).
- `branch`: initial branch (see table 9).
- `point_numbers` (optional): vector of one or more point numbers which should be recomputed. Empty or absent if the complete point array should be recomputed.
- `recmp_branch`: a copy of the initial branch with points who were (successfully) recomputed replaced. If a recomputation fails, a warning message is given and the old value remains present.

This routine can, e.g., be used after changing some method parameters within the branch method structures.

```
function [col,lengths]=br_measr(branch,measure)
```

Function `br_selec` computes a measure along a branch.

- `branch`: given branch (see table 9).
- `measure`: given measure (see table 10).
- `col`: the collection of measures taken along the branch (over its point array) ordered row-wise. Thus, a column vector is returned if `measure` is scalar. Otherwise, `col` contains a matrix.
- `lengths`: vector of lengths of the measures along the branch. If the measure is not scalar, it is possible that its length varies along the branch (e.g. when plotting rightmost characteristic roots). In this situation `col` is a matrix with number of columns equal to the maximal length of the measures encountered. Extra elements of `col` are automatically put to zero by Matlab. `lengths` can then be used to prevent plotting of extra zeros.

10. Numerical methods

This section contains short descriptions of the numerical methods for DDEs and the method parameters used in DDE-BIFTOOL. More details on the methods can be found in the articles [35, 18, 17, 16, 19, 39] or in [15]. For details on applying these methods to bifurcation analysis of sd-DDEs see [34].

10.1. Determining systems

Below we state the determining systems used to compute and continue steady state solutions, steady state fold and Hopf bifurcations, periodic solutions and connecting orbits of systems of delay differential equations.

For each determining system we mention the number of free parameters necessary to obtain (generically) isolated solutions. In the package, the necessary number of free parameters is further raised by the number of steplength conditions plus the number of extra conditions used. This choice ensures the use of square Jacobians during Newton iteration. If, on the other hand, the number of free parameters, steplength conditions and extra conditions are not appropriately matched Newton iteration solves systems with a non-square Jacobian (for which Matlab uses an over- or under-determined least squares procedure). If possible, it is better to avoid such a situation.

Steady state solutions A steady state solution $x^* \in \mathbb{R}^n$ is determined from the following n-dimensional determining system with no free parameters.

$$f(x^*, x^*, \ldots, x^*, \eta) = 0. \tag{12}$$

Steady state fold bifurcations Fold bifurcations, $(x^* \in \mathbb{R}^n, v \in \mathbb{R}^n)$ are determined from the following $2n+1$-dimensional determining system using one free parameter.

$$\begin{aligned} 0 &= f(x^*, x^*, \ldots, x^*, \eta) \\ 0 &= \Delta(x^*, \eta, 0)v \\ 0 &= c^{\mathrm{T}}v - 1 \end{aligned} \tag{13}$$

(see (4) for the definition of the characteristic matrix Δ). Here, $c^{\mathrm{T}}v - 1 = 0$ presents a suitable normalization of v. The vector $c \in \mathbb{R}^n$ is chosen as $c = v^{(0)}/({v^{(0)}}^{\mathrm{T}}v^{(0)})$, where $v^{(0)}$ is the initial value of v.

Steady state Hopf bifurcations Hopf bifurcations, $(x^* \in \mathbb{R}^n, v \in \mathbb{C}^n, \omega \in \mathbb{R})$ are determined from the following $2n+1$-dimensional partially complex (and by this fact more properly called a $3n+2$-dimensional) determining system using one free parameter.

$$\begin{aligned} 0 &= f(x^*, x^*, \ldots, x^*, \eta) \\ 0 &= \Delta(x^*, \eta, \mathrm{i}\omega)v \\ 0 &= c^{\mathrm{H}}v - 1 \end{aligned} \tag{14}$$

Periodic solutions Periodic solutions are found as solutions $(u(s),\ s \in [0,1]; T \in \mathbb{R})$ of the following $(n(Ld+1)+1$-dimensional system with no free parameters.

$$\begin{aligned}
\dot{u}(c_{i,j}) &= Tf\left(u(c_{i,j}), u\left(\left[c_{i,j} - \frac{\tau_1}{T}\right|_{\text{mod }[0,1]}\right), \dots, u\left(\left[c_{i,j} - \frac{\tau_m}{T}\right|_{\text{mod }[0,1]}\right)\right), \\
&\qquad i = 0, \dots, L-1,\ j = 1, \dots, d \\
u(0) &= u(1), \\
p(u) &= 0.
\end{aligned} \tag{15}$$

Here the notation $t|_{\text{mod }[0,1]}$ refers to $t - \max\{k \in \mathbb{Z} : k \le t\}$, and p represents the integral phase condition

$$\int_0^1 \dot{u}(s)\Delta u(s)\mathrm{d}s = 0, \tag{16}$$

where u is the current solution and Δu its correction. The collocation points are obtained as

$$c_{i,j} = t_i + c_j(t_{i+1} - t_i),\ i = 0, \dots, L-1,\ j = 1, \dots, d,$$

from the interval points t_i, $i = 0, \dots, L-1$ and the collocation parameters c_j, $j = 1, \dots, d$. The profile u is discretized as a piecewise polynomial as explained in section 7. This representation has a discontinuous derivative at the interval points. If $c_{i,j}$ coincides with t_i the right derivative is taken in (15), if it coincides with t_{i+1} the left derivative is taken. In other words the derivative taken at $c_{i,j}$ is that of u restricted to $[t_i, t_{i+1}]$.

Connecting orbits Connecting orbits can be found as solutions of the following determining system with $s^+ - s^- + 1$ free parameters, where s^+ and s^- denote the number of unstable eigenvalues of x^+ and x^- respectively.

$$\begin{aligned}
\dot{u}(c_{i,j}) &= Tf(u(c_{i,j}), u(c_{i,j} - \frac{\tau_1}{T}), \dots, u(c_{i,j} - \frac{\tau_m}{T}), \eta) = 0, \quad (i = 0, \dots, L-1,\ j = 1, \dots, d) \\
u(\tilde{c}) &= x^- + \epsilon \sum_{k=1}^{s^-} \alpha_k v_k^- e^{\lambda_k^- T\tilde{c}}, \qquad \tilde{c} < 0 \\
0 &= f(x^-, x^-, \eta) \\
0 &= f(x^+, x^+, \eta) \\
0 &= \Delta(x^-, \lambda_k^-, \eta)v_k^- \\
0 &= c_k^H v_k^- - 1, \quad (k = 1, \dots, s^-) \\
0 &= \Delta^H(x^+, \lambda_k^+, \eta)w_k^+ \\
0 &= d_k^H w_k^+ - 1, \quad (k = 1, \dots, s^+) \\
0 &= w_k^{2^H}(u(1) - x^+) + \sum_{i=1}^{G} g_i w_k^{+^H} e^{-\lambda_k^+(\theta_i + \tau)} A_1(x^+, \eta)\left(u(1 + \frac{\theta_i}{T}) - x^+\right), \quad (k = 1, \dots, s^+) \\
u(0) &= x^- + \epsilon \sum_{i=1}^{s^-} \alpha_k v_k^- \\
1 &= \sum_{i=1}^{s^-} |\alpha_k|^2 \\
0 &= p(u, \eta)
\end{aligned} \tag{17}$$

Again, all arguments of u are taken modulo $[0,1]$. We choose the same phase condition as for periodic solutions and similar normalization of v_k^- and $w + k^+$ as in (14).

Point method parameters The point method parameters (see table 6) specify the following options.

- `newton_max_iterations`: maximum number of Newton iterations.
- `newton_nmon_iterations`: during a first phase of `newton_nmon_iterations`+1 Newton iterations the norm of the residual is allowed to increase. After these iterations, corrections are halted upon residual increase.
- `halting_accuracy`: corrections are halted when the norm of the last computed residual is less than or equal to `halting_accuracy` is reached.
- `minimal_accuracy`: a corrected point is accepted when the norm of the last computed residual is less than or equal to `minimal_accuracy`.
- `extra_condition`: this parameter is nonzero when extra conditions are provided in a routine `sys_cond.m` which should border the determining systems during corrections. The routine accepts the current point as input and produces an array of condition residuals and corresponding condition derivatives (as an array of point structures) as illustrated below (§10.2).
- `print_residual_info`: when nonzero, the Newton iteration number and resulting norm of the residual are printed to the screen during corrections.

For periodic solutions and connecting orbits, the extra mesh parameters (see table 6) provide the following information.

- `phase_condition`: when nonzero the integral phase condition (16) is used.
- `collocation_parameters`: this parameter contains user given collocation parameters. When empty, Gauss-Legendre collocation points are chosen.
- `adapt_mesh_before_correct`: before correction and if the mesh inside the point is nonempty, adapt the mesh every `adapt_mesh_before_correct` points. E.g.: if zero, do not adapt; if one, adapt every point; if two adapt the points with odd point number.
- `adapt_mesh_after_correct`: similar to `adapt_mesh_before_correct` but adapt mesh after successful corrections and correct again.

10.2. Extra conditions

When correcting a point or computing a branch, it is possible to add one or more extra conditions and corresponding free parameters to the determining systems presented earlier. These extra conditions should be implemented using a function `sys_cond` and setting the method parameter `extra_condition` to `1` (cf. table 6). The function `sys_cond` accepts the current point as input and produces a residual and corresponding condition derivatives (as a point structure) per extra condition.

As an example, suppose we want to compute a branch of periodic solutions of system (10) subject to the following extra conditions

$$\begin{aligned} T &= 200, \\ 0 &= a_{12}^2 + a_{21}^2 - 1, \end{aligned} \tag{18}$$

that is, we wish to continue a branch with fixed period $T = 200$ and parameter dependence $a_{12}^2 + a_{21}^2 = 1$. The routine shown in Listing 5 implements these conditions by evaluating and returning each residual for the given point and the derivatives of the conditions w.r.t. all unknowns (that is, w.r.t. to all the components of the point structure).

```
function [resi,condi]=sys_cond(point)
% kappa beta a12 a21 tau1 tau2 tau_s
if point.kind=='psol'
  % fix period at 200:
  resi(1)=point.period-200;
  % derivative of first condition wrt unknowns:
  condi(1)=p_axpy(0,point,[]);
  condi(1).period=1;
  % parameter condition:
  resi(2)=point.parameter(3)^2+point.parameter(4)^2-1;
  % derivative of second condition wrt unknowns:
  condi(2)=p_axpy(0,point,[]);
  condi(2).parameter(3)=2*point.parameter(3);
  condi(2).parameter(4)=2*point.parameter(4);
else
  error('SYS_COND: point is not psol.');
end
end
```

Listing 5: Implementation extra conditions (18) using a routine sys_cond.

10.3. Continuation

During continuation, a branch is extended by a combination of predictions and corrections. A new point is predicted based on previously computed points using secant prediction over an appropriate steplength. The prediction is then corrected using the determining systems (12), (13), (14), (15) or (17) bordered with a steplength condition which requires orthogonality of the correction to the secant vector. Hence one extra free parameter is necessary compared to the numbers mentioned in the previous section.

The following continuation and steplength determination strategy is used. If the last point was successfully computed, the steplength is multiplied with a given, constant factor greater than 1. If corrections diverged or if the corrected point was rejected because its accuracy was not acceptable, a new point is predicted, using linear interpolation, halfway between the last two successfully computed branch points. If the correction of this point succeeds, it is inserted in the point array of the branch (before the previously last computed point). If the correction of the interpolated point fails again, the last successfully computed branch point is rejected (for fear of branch switch) and the interpolation procedure is repeated between the (new) last two branch points. Hence, if, after a failure, the interpolation procedure succeeds, the steplength is approximately divided by a factor two. Test results indicate that this procedure is quite effective and proves an efficient alternative to using only (secant) extrapolation with steplength control. The reason for this is mainly that the secant extrapolation direction is not influenced by halving the steplength but it is by inserting a newly computed point in between the last two computed points.

Continuation method parameters The continuation method parameters (see table 8) have the following meaning.

- plot: if nonzero, plot predictions and corrections during continuation.
- prediction: this parameter should be 1, indicating that secant prediction is used (being currently the only alternative).

- `steplength_growth_factor`: grow the steplength with this factor in every step except during interpolation.
- `plot_progress`: if nonzero, plotting is visible during continuation process. If zero, only the final result is drawn.
- `plot_measure`: if empty use default measures to plot. Otherwise `plot_measure` contains two fields, 'x' and 'y', which contain measures (see table 10) for use in plotting during continuation.
- `halt_before_reject`: If this parameter is nonzero, continuation is halted whenever (and instead of) rejecting a previously accepted point based on the above strategy.

10.4. Roots of the characteristic equation

Roots of the characteristic equation are approximated using a linear multi-step (LMS-) method applied to (2).

Consider the linear k-step formula

$$\sum_{j=0}^{k} \alpha_j y_{L+j} = h \sum_{j=0}^{k} \beta_j f_{L+j}. \tag{19}$$

Here, $\alpha_0 = 1$, h is a (fixed) step size and y_j presents the numerical approximation of $y(t)$ at the mesh point $t_j := jh$. The right hand side $f_j := f(y_j, \tilde{y}(t_j - \tau_1), \ldots, \tilde{y}(t_j - \tau_m))$ is computed using approximations $\tilde{y}(t_j - \tau_1)$ obtained from y_i in the past, $i < j$. In particular, the use of so-called Nordsieck interpolation, leads to

$$\tilde{y}(t_j + \epsilon h) = \sum_{l=-r}^{s} P_l(\epsilon) y_{j+l},\ \epsilon \in [0, 1). \tag{20}$$

using

$$P_l(\epsilon) := \prod_{k=-r, k\neq l}^{s} \frac{\epsilon - k}{l - k}.$$

The resulting method is explicit whenever $\beta_0 = 0$ and $\min \tau_i > sh$. That is, y_{L+k} can then directly be computed from (19) by evaluating

$$y_{L+k} = -\sum_{j=0}^{k-1} \alpha_j y_{L+j} + h \sum_{j=0}^{k} \beta_j f_{L+j}.$$

whose right hand side depends only on y_j, $j < L + k$.

For the linear variational equation (2) around a steady state solution $x^*(t) \equiv x^*$ we have

$$f_j = A_0 y_j + \sum_{i=0}^{m} A_i \tilde{y}(t_j - \tau_i) \tag{21}$$

where we have omitted the dependency of A_i on x^*. The stability of the difference scheme (19), (21) can be evaluated by setting $y_j = \mu^{j - L_{\min}}$, $j = L_{\min}, \ldots, L + k$ where $L_{\min}$ is the smallest index used, taking the determinant of (19) and computing the roots μ. If the roots of the polynomial in μ all have modulus smaller than unity, the trajectories of the LMS-method converge to zero. If roots exist with modulus greater than unity then trajectories exist which grow unbounded.

Since the LMS-method forms an approximation of the time integration operator over the time step h, so do the roots μ approximate the eigenvalues of $S(h, 0)$. The eigenvalues of $S(h, 0)$ are exponential transforms of the roots λ of the characteristic equation (5),

$$\mu = \exp(\lambda h).$$

Hence, once μ is found, λ can be extracted using,

$$\mathrm{Re}(\lambda) = \frac{\ln(|\mu|)}{h}. \tag{22}$$

The imaginary part of λ is found modulo π/h, using

$$\mathrm{Im}(\lambda) \equiv \frac{\arcsin(\frac{\mathrm{Im}(\mu)}{|\mu|})}{h} \pmod{\frac{\pi}{h}}. \tag{23}$$

For small h, $0 < h \ll 1$, the smallest representation in (23) is assumed the most accurate one (that is, we let arcsin map into $[-\pi/2, \pi/2]$).

The parameters r and s (from formula (20)) are chosen such that $r \leq s \leq r+2$ (see [28]). The choice of h is based on the related heuristic outlined in [19].

Approximations for the rightmost roots λ obtained from the LMS-method using (22), (23) can be corrected using a Newton process on the system,

$$\begin{aligned} 0 &= \Delta(\lambda)v \\ 0 &= c^{\mathrm{T}}v - 1 \end{aligned} \tag{24}$$

A starting value for v is the eigenvector of $\Delta(\lambda)$ corresponding to its smallest eigenvalue (in modulus).

Note that the collection of successfully corrected roots presents more accurate yet less robust information than the set of uncorrected roots. Indeed, attraction domains of roots of equations like (24) can be very small and hence corrections may diverge, or different roots may be corrected to a single 'exact' root thereby missing part of the spectrum. The latter does not occur when computing the (full) spectrum of a discretization of $S(h, 0)$.

Stability information is kept in the structure of table 5 (left). The time step used is kept in field `h`. Approximate roots are kept in field `l0`, corrected roots in field `l1`. If unconverged corrected roots are discarded, field `n1` is empty. Otherwise, the number of Newton iterations used is kept for each root in the corresponding position of `n1`. Here, -1 signals that convergence to the required accuracy was not reached.

Stability method parameters The stability method parameters (see table 7 (top)) now have the following meaning.

- `lms_parameter_alpha`: LMS-method parameters α_j ordered from past to present, $j = 0, 1, \ldots, k$.
- `lms_parameter_beta`: LMS-method parameters β_j ordered from past to present, $j = 0, 1, \ldots, k$.
- `lms_parameter_rho`: safety radius $\rho_{\mathrm{LMS},\epsilon}$ of the LMS-method stability region. For a precise definition, see [15, §III.3.2].
- `interpolation_order`: order of the interpolation in the past, $r + s =$ `interpolation_order`.
- `minimal_time_step`: minimal time step relative to maximal delay, $\frac{h}{\tau} \geq$ `minimal_time_step`.
- `maximal_time_step`: maximal time step relative to maximal delay, $\frac{h}{\tau} \leq$ `minimal_time_step`.
- `max_number_of_eigenvalues`: maximum number of rightmost eigenvalues to keep.
- `minimal_real_part`: choose h such as the discretized system approximates eigenvalues with $\mathrm{Re}(\lambda) \geq$ `minimal_real_part` well, discard eigenvalues with $\mathrm{Re}(\lambda) <$ `minimal_real_part`. If h is smaller than its minimal value, it is set to the minimal value and a warning is given. If it is larger than its maximal value it is reduced to that number without warning. If minimal and maximal value coincide, h is set to this value without warning. If `minimal_real_part` is empty, the value `minimal_real_part` $= \frac{1}{\tau}$ is used.

- `max_newton_iterations`: maximum number of Newton iterations during the correction process (24).
- `root_accuracy`: required accuracy of the norm of the residual of (24) during corrections.
- `remove_unconverged_roots`: if this parameter is zero, unconverged roots are discarded (and stability field `n1` is empty).
- `delay_accuracy` (only for state-dependent delays): if the value of a state-dependent delay is less than `delay_accuracy`, the stability is not computed.

10.5. Floquet multipliers

Floquet multipliers are computed as eigenvalues of the discretized time integration operator $S(T, 0)$. The discretization is obtained using the collocation equations (15) without the modulo operation (and without phase and periodicity condition). From this system a discrete, linear map is obtained between the variables presenting the segment $[-\tau/T, 0]$ and those presenting the segment $[-\tau/T + 1, 1]$. If these variables overlap, part of the map is just a time shift.

Stability information is kept in the structure of table 5 (right). Approximations to the Floquet multipliers are kept in field `mu`.

Stability method parameters The stability method parameters (see table 7 (bottom)) have the following meaning.

- `collocation_parameters`: user given collocation parameters or empty for Gauss-Legendre collocation points.
- `max_number_of_eigenvalues`: maximum number of multipliers to keep.
- `minimal_modulus`: discard multipliers with $|\mu| <$ `minimal_modulus`.
- `delay_accuracy` (only for state-dependent delays): if the value of a state-dependent delay is less than `delay_accuracy`, the stability is not computed.

11. Concluding comments

The first aim of DDE-BIFTOOL is to provide a portable, user-friendly tool for numerical bifurcation analysis of steady state solutions and periodic solutions of systems of delay differential equations of the kinds (1) and (7). Part of this goal was fulfilled through choosing the portable, programmer-friendly environment offered by Matlab. Robustness with respect to the numerical approximation is achieved through automatic steplength selection in approximating the rightmost characteristic roots and through collocation using piecewise polynomials combined with adaptive mesh selection.

Although the package has been successfully tested on a number of realistic examples, a word of caution may be appropriate. First of all, the package is essentially a research code (hence we accept no reliability) in a quite unexplored area of current research. In our experience up to now, new examples did not fail to produce interesting theoretical questions (e.g., concerning homoclinic or heteroclinic solutions) many of which remain unsolved today. Unlike for ordinary differential equations, discretization of the state space is unavoidable during computations on delay equations. Hence the user of the package is strongly advised to investigate the effect of discretization using tests on different meshes and with different method parameters; and, if possible, to compare with analytical results and/or results obtained using simulation.

Although there are no 'hard' limits programmed in the package (with respect to system and/or mesh sizes), the user will notice the rapidly increasing computation time for increasing system dimension and mesh sizes. This is most notable in the stability and periodic solution computations. Indeed, eigenvalues are computed from large sparse matrices without exploiting sparseness and the Newton procedure for periodic solutions is implemented using direct methods. Nevertheless the current version is sufficient to perform bifurcation analysis of systems with reasonable properties in reasonable execution times. Furthermore, we hope future versions will include routines which scale better with the size of the problem.

11.1. Existing extensions

- Extension `debiftool_extra_psol` continues the three local codimension-one bifurcations of periodic orbits, the fold bifurcation, the period doubling and the torus bifurcation for constant and state-dependent delays.
- Extension `debiftool_extra_rotsym` continues relative equilibria and relative periodic orbits and their local codimension-one bifurcations for constant delays in systems with rotational symmetry (that is, there exists a matrix $A \in \mathbb{R}^{n \times n}$ such that $A^T = -A$ and $\exp(At) f(x_0, \dots, x_m) = f(\exp(At)x_0, \dots, \exp(At)x_m)$).
- Extension `debiftool_extra_nmfm` computes normal form coefficients of Hopf bifurcations, Hopf-Hopf interactions, generalized Hopf (Bautin) bifurcations, and zero-Hopf interactions (Gavrilov-Guckenheimer bifurcations) for equations with constant delays.

Other possible future developments include

- a graphical user interface;
- incorporation of the numerical core routines into a general continuation framework such as `Coco` [8] (which would permit the user to grow higher-dimensional solution families and wrap other continuation algorithms around the core DDE routines),
- the extension to other types of delay equations such as distributed delay and neutral functional differential equations. See also Barton *et al* [3] for a demonstration of how to extend DDE-BIFTOOL to neutral functional differential equations.
- determination of more normal-form coefficients to detect other co-dimension-two bifurcations.

Acknowledgements

DDE-BIFTOOL v. 2.03 is a result of the research project OT/98/16, funded by the Research Council K.U.Leuven; of the research project G.0270.00 funded by the Fund for Scientific Research - Flanders (Belgium) and of the research project IUAP P4/02 funded by the programme on Interuniversity Poles of Attraction, initiated by the Belgian State, Prime Minister's Office for Science, Technology and Culture. K. Engelborghs is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium). J. Sieber's contribution to the revision leading to version 3.0 was supported by EPSRC grant EP/J010820/1.

References

[1] J. Argyris, G. Faust, and M. Haase. *An Exploration of Chaos — An Introduction for Natural Scientists and Engineers*. North Holland Amsterdam, 1994.

[2] N. V. Azbelev, V. P. Maksimov, and L. F. Rakhmatullina. *Introduction to the Theory of Functional Differential Equations*. Nauka, Moscow, 1991. (in Russian).

[3] D.A.W. Barton, B. Krauskopf, and R.E. Wilson. Collocation schemes for periodic solutions of neutral delay differential equations. *Journal of Difference Equations and Applications*, 12(11):1087–1101, 2006.

[4] R. Bellman and K. L. Cooke. *Differential-Difference Equations*, volume 6 of *Mathematics in science and engineering*. Academic Press, 1963.

[5] D. Breda, S. Maset, and R. Vermiglio. TRACE-DDE: a tool for robust analysis and characteristic equations for delay differential equations. In J.J. Loiseau, W. Michiels, S.-I. Niculescu, and R. Sipahi, editors, *Topics in Time Delay Systems: Analysis, Algorithms, and Control*, volume 388 of *Lecture Notes in Control and Information Sciences*, pages 145–155, New York, 2009. Springer.

[6] S.-N. Chow and J. K. Hale. *Methods of Bifurcation Theory*. Springer-Verlag, 1982.

[7] S. P. Corwin, D. Sarafyan, and S. Thompson. DKLAG6: A code based on continuously imbedded sixth order Runge-Kutta methods for the solution of state dependent functional differential equations. *Applied Numerical Mathematics*, 24(2–3):319–330, 1997.

[8] H. Dankowicz and F. Schilder. *Recipes for Continuation*. Computer Science and Engineering. SIAM, 2013.

[9] A Dhooge, W Govaerts, and Y A Kuznetsov. MatCont: A Matlab package for numerical bifurcation analysis of ODEs. *ACM Transactions on Mathematical Software*, 29(2):141–164, 2003.

[10] O. Diekmann, S. A. van Gils, S. M. Verduyn Lunel, and H.-O. Walther. *Delay Equations: Functional-, Complex-, and Nonlinear Analysis*, volume 110 of *Applied Mathematical Sciences*. Springer-Verlag, 1995.

[11] E J Doedel. Lecture notes on numerical analysis of nonlinear equations. In B Krauskopf, H M Osinga, and J Galán-Vioque, editors, *Numerical Continuation Methods for Dynamical Systems: Path following and boundary value problems*, pages 1–49. Springer-Verlag, Dordrecht, 2007.

[12] E. J. Doedel, A. R. Champneys, T. F. Fairgrieve, Y. A. Kuznetsov, B. Sandstede, and X. Wang. *AUTO97: Continuation and bifurcation software for ordinary differential equations; available by FTP from ftp.cs.concordia.ca in directory pub/doedel/auto.*

[13] R. D. Driver. *Ordinary and Delay Differential Equations*, volume 20 of *Applied Mathematical Science*. Springer-Verlag, 1977.

[14] L. E. El'sgol'ts and S. B. Norkin. *Introduction to the Theory and Application of Differential Equations with Deviating Arguments*, volume 105 of *Mathematics in science and engineering*. Academic Press, 1973.

[15] K. Engelborghs. *Numerical Bifurcation Analysis of Delay Differential Equations*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium, May 2000.

[16] K. Engelborghs and E. Doedel. Stability of piecewise polynomial collocation for computing periodic solutions of delay differential equations. *Numerische Mathematik*, 2001. Accepted.

[17] K. Engelborghs, T. Luzyanina, K. J. in ´t Hout, and D. Roose. Collocation methods for the computation of periodic solutions of delay differential equations. *SIAM J. Sci. Comput.*, 22:1593–1609, 2000.

[18] K. Engelborghs and D. Roose. 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, 1999.

[19] K. Engelborghs and D. Roose. On stability of LMS-methods and characteristic roots of delay differential equations. *SIAM J. Num. Analysis*, 2001. Accepted.

[20] W. H. Enright and H. Hayashi. A delay differential equation solver based on a continuous Runge-Kutta method with defect control. *Numer. Algorithms*, 16:349–364, 1997.

[21] B. Ermentrout. *XPPAUT3.91 - The differential equations tool.* University of Pittsburgh, Pittsburgh, (http://www.pitt.edu/∼phase/) 1998.

[22] W.J.F. Govaerts. *Numerical Methods for Bifurcations of Dynamical Equilibria*. Miscellaneous Titles in Applied Mathematics Series. SIAM, 2000.

[23] J. Guckenheimer and P. Holmes. *Nonlinear Oscillations, Dynamical Systems and Bifurcations of Vector Fields*. Springer-Verlag New York, 1983.

[24] N. Guglielmi and E. Hairer. Stiff delay equations. *Scholarpedia*, 2(11):2850, 2007.

[25] J. K. Hale. *Theory of Functional Differential Equations*, volume 3 of *Applied Mathematical Sciences*. Springer-Verlag, 1977.

[26] J. K. Hale and S. M. Verduyn Lunel. *Introduction to Functional Differential Equations*, volume 99 of *Applied Mathematical Sciences*. Springer-Verlag, 1993.

[27] F. Hartung, T. Krisztin, H.-O. Walther, and J. Wu. Functional differential equations with state-dependent delays: theory and applications. In P. Drábek, A. Cañada, and A. Fonda, editors, *Handbook of Differential Equations*: *Ordinary Differential Equations*, volume 3, chapter 5, pages 435–545. North-Holland, 2006.

[28] T. Hong-Jiong and K. Jiao-Xun. The numerical stability of linear multistep methods for delay differential equations with many delays. *SIAM Journal of Numerical Analysis*, 33(3):883–889, June 1996.

[29] The MathWorks Inc. MATLAB. Natick, Massachusetts, United States.

[30] V. Kolmanovskii and A. Myshkis. *Applied Theory of Functional Differential Equations*, volume 85 of *Mathematics and Its Applications*. Kluwer Academic Publishers, 1992.

[31] V. B. Kolmanovskii and A. Myshkis. *Introduction to the theory and application of functional differential equations*, volume 463 of *Mathematics and its applications*. Kluwer Academic Publishers, 1999.

[32] V. B. Kolmanovskii and V. R. Nosov. *Stability of functional differential equations*, volume 180 of *Mathematics in Science and Engineering*. Academic Press, 1986.

[33] Y. A Kuznetsov. *Elements of Applied Bifurcation Theory*, volume 112 of *Applied Mathematical Sciences*. Springer-Verlag, New York, third edition, 2004.

[34] T. Luzyanina, K. Engelborghs, and D. Roose. Numerical bifurcation analysis of differential equations with state-dependent delay. *Internat. J. Bifur. Chaos*, 11(3):737–754, 2001.

[35] T. Luzyanina and D. Roose. Numerical stability analysis and computation of Hopf bifurcation points for delay differential equations. *Journal of Computational and Applied Mathematics*, 72:379–392, 1996.

[36] J. Mallet-Paret and R. D. Nussbaum. Stability of periodic solutions of state-dependent delay-differential equations. *Journal of Differential Equations*, 250(11):4085 – 4103, 2011.

[37] C. A. H. Paul. A user-guide to Archi - an explicit Runge-Kutta code for solving delay and neutral differential equations. Technical Report 283, The University of Manchester, Manchester Center for Computational Mathematics, December 1997.

[38] D. Roose and R. Szalai. Continuation and bifurcation analysis of delay differential equations. In B Krauskopf, H M Osinga, and J Galán-Vioque, editors, *Numerical Continuation Methods for Dynamical Systems: Path following and boundary value problems*, pages 51–75. Springer-Verlag, Dordrecht, 2007.

[39] G. Samaey, K. Engelborghs, and D. Roose. Numerical computation of connecting orbits in delay differential equations. Report TW 329, Department of Computer Science, K.U.Leuven, Leuven, Belgium, October 2001.

[40] R. Seydel. *Practical Bifurcation and Stability Analysis — From Equilibrium to Chaos*, volume 5 of *Interdisciplinary Applied Mathematics*. Springer-Verlag Berlin, 2 edition, 1994.

[41] L. F. Shampine and S. Thompson. Solving delay differential equations with dde23. Submitted, 2000.

[42] L. P. Shayer and S. A. Campbell. Stability, bifurcation and multistability in a system of two coupled neurons with multiple time delays. *SIAM J. Applied Mathematics*, 61(2):673–700, 2000.

[43] J. Sieber. Finding periodic orbits in state-dependent delay differential equations as roots of algebraic equations. *Discrete and Continuous Dynamical Systems A*, 32(8):2607–2651, 2012.

[44] R. Szalai, G. Stépán, and S.J. Hogan. Continuation of bifurcations in periodic delay differential equations using characteristic matrices. *SIAM Journal on Scientific Computing*, 28(4):1301–1317, 2006.

[45] K. Verheyden, T. Luzyanina, and D. Roose. Efficient computation of characteristic roots of delay differential equations using lms methods,. *Journal of Computational and Applied Mathematics*, 214:209–226, 2008.

[46] Bram Wage. Normal form computations for delay differential equations in DDE-BIFTOOL. Master's thesis, Utrecht University, 2014. supervised by Y.A. Kuznetsov.

A. Jacobians of tutorial examples `neuron` and `sd_demo`

The meaning of input arguments to `sys_deri` is explained in section 6.2.3. The analysis of tutorial example `neuron` is demonstrated in demo `neuron` step by step (see `../demos/index.html`).

```
function J=neuron_sys_deri(xx,par,nx,np,v)
%% Parameter vector
% par=[\kappa, \beta, a_{12}, a_{21},\tau_1,\tau_2, \tau_s].
J=[];
if length(nx)==1 && isempty(np) && isempty(v)
    %% First-order derivatives wrt to state nx+1
    if nx==0 % derivative wrt x(t)
        J(1,1)=-par(1);
        J(2,2)=-par(1);
    elseif nx==1 % derivative wrt x(t-tau1)
        J(2,1)=par(4)*(1-tanh(xx(1,2))^2);
        J(2,2)=0;
    elseif nx==2 % derivative wrt x(t-tau2)
        J(1,2)=par(3)*(1-tanh(xx(2,3))^2);
        J(2,2)=0;
    elseif nx==3 % derivative wrt x(t-tau_s)
        J(1,1)=par(2)*(1-tanh(xx(1,4))^2);
        J(2,2)=par(2)*(1-tanh(xx(2,4))^2);
    end
elseif isempty(nx) && length(np)==1 && isempty(v)
    %% First order derivatives wrt parameters
    if np==1 % derivative wrt kappa
        J(1,1)=-xx(1,1);
        J(2,1)=-xx(2,1);
    elseif np==2 % derivative wrt beta
        J(1,1)=tanh(xx(1,4));
        J(2,1)=tanh(xx(2,4));
    elseif np==3 % derivative wrt a12
        J(1,1)=tanh(xx(2,3));
        J(2,1)=0;
    elseif np==4 % derivative wrt a21
        J(2,1)=tanh(xx(1,2));
    elseif np==5 || np==6 || np==7 % derivative wrt tau
        J=zeros(2,1);
    end;
elseif length(nx)==1 && length(np)==1 && isempty(v)
    %% Mixed state, parameter derivatives
    if nx==0 % derivative wrt x(t)
        if np==1 % derivative wrt beta
            J(1,1)=-1;
            J(2,2)=-1;
        else
            J=zeros(2);
        end;
    elseif nx==1 % derivative wrt x(t-tau1)
        if np==4 % derivative wrt a21
            J(2,1)=1-tanh(xx(1,2))^2;
            J(2,2)=0;
        else
            J=zeros(2);
        end;
    elseif nx==2 % derivative wrt x(t-tau2)
        if np==3 % derivative wrt a12
            J(1,2)=1-tanh(xx(2,3))^2;
            J(2,2)=0;
        else
            J=zeros(2);
        end;
    elseif nx==3 % derivative wrt x(t-tau_s)
        if np==2 % derivative wrt beta
            J(1,1)=1-tanh(xx(1,4))^2;
            J(2,2)=1-tanh(xx(2,4))^2;
        else
            J=zeros(2);
        end;
    end;
elseif length(nx)==2 && isempty(np) && ~isempty(v)
    %% Second order derivatives wrt state variables
    if nx(1)==0 % first derivative wrt x(t)
        J=zeros(2);
    elseif nx(1)==1 % first derivative wrt x(t-tau1)
        if nx(2)==1
            th=tanh(xx(1,2));
            J(2,1)=-2*par(4)*th*(1-th*th)*v(1);
            J(2,2)=0;
        else
            J=zeros(2);
        end;
    elseif nx(1)==2 % derivative wrt x(t-tau2)
        if nx(2)==2
            th=tanh(xx(2,3));
            J(1,2)=-2*par(3)*th*(1-th*th)*v(2);
            J(2,2)=0;
        else
            J=zeros(2);
        end
    elseif nx(1)==3 % derivative wrt x(t-tau_s)
        if nx(2)==3
            th1=tanh(xx(1,4));
            J(1,1)=-2*par(2)*th1*(1-th1*th1)*v(1);
            th2=tanh(xx(2,4));
            J(2,2)=-2*par(2)*th2*(1-th2*th2)*v(2);
        else
            J=zeros(2);
        end
    end
end

% Raise error if the requested derivative does not exist
```

```
if isempty(J)
    error(['SYS_DERI: requested derivative nx=%d, np=%d,',...
        ' size(v)=%d could not be computed!'],nx,np,size(v));
end
end
```

Listing 6: Jacobians of right-hand side `neuron_sys_rhs` in section 6.2.1 for `neuron_sys_rhs`.

The meaning of input arguments to `sys_dtau` is explained in section 6.3.4. The analysis of tutorial example `sd_demo` is demonstrated step by step in demo `sd_demo` (see `../demos/index.html`).

```
function dtau=sd_dtau(ind,xx,par,nx,np)
% p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11
dtau=[];
if length(nx)==1 && isempty(np)
    %% first order derivatives wrt state variables:
    if nx==0 % derivative wrt x(t)
        if ind==3
            dtau(1:5)=0;
            dtau(2)=par(5)*par(10)*xx(2,2);
        elseif ind==4
            dtau(1)=xx(2,3)/(1+xx(1,1)*xx(2,3))^2;
            dtau(2:5)=0;
        elseif ind==5
            dtau(1:5)=0;
            dtau(4)=1;
        elseif ind==6
            dtau(5)=1;
        else
            dtau(1:5)=0;
        end;
    elseif nx==1 % derivative wrt x(t-tau1)
        if ind==3
            dtau(1:5)=0;
            dtau(2)=par(5)*par(10)*xx(2,1);
        else
            dtau(1:5)=0;
        end;
    elseif nx==2 % derivative wrt x(t-tau2)
        if ind==4
            dtau(1:5)=0;
            dtau(2)=xx(1,1)/(1+xx(1,1)*xx(2,3))^2;
        else
            dtau(1:5)=0;
        end;
    else
        dtau(1:5)=0;
    end;
elseif isempty(nx) && length(np)==1,
    %% First order derivatives wrt parameters
    if ind==1 && np==10
        dtau=1;
    elseif ind==2 && np==11
```

```
        dtau=1;
    elseif ind==3 && np==5
        dtau=par(10)*xx(2,1)*xx(2,2);
    elseif ind==3 && np==10
        dtau=par(5)*xx(2,1)*xx(2,2);
    else
        dtau=0;
    end;
elseif length(nx)==2 && isempty(np),
    %% Second order derivatives wrt state variables
    dtau=zeros(5);
    if ind==3
        if (nx(1)==0 && nx(2)==1) || (nx(1)==1 && nx(2)==0)
            dtau(2,2)=par(5)*par(10);
        end
    elseif ind==4
        if nx(1)==0 && nx(2)==0
            dtau(1,1)=-2*xx(2,3)*xx(2,3)/(1+xx(1,1)*xx(2,2))^3;
        elseif nx(1)==0 && nx(2)==2
            dtau(1,2)=(1-xx(1,1)*xx(2,3))/(1+xx(1,1)*xx(2,2))^3;
        elseif nx(1)==2 && nx(2)==0
            dtau(2,1)=(1-xx(1,1)*xx(2,3))/(1+xx(1,1)*xx(2,2))^3;
        elseif nx(1)==2 && nx(2)==2
            dtau(2,2)=-2*xx(1,1)*xx(1,1)/(1+xx(1,1)*xx(2,2))^3;
        end
    end
elseif length(nx)==1 && length(np)==1,
    %% Mixed state parameter derivatives
    dtau(1:5)=0;
    if ind==3
        if nx==0 && np==5
            dtau(2)=par(10)*xx(2,2);
        elseif nx==0 && np==10
            dtau(2)=par(5)*xx(2,2);
        elseif nx==1 && np==5
            dtau(2)=par(10)*xx(2,1);
        elseif nx==1 && np==10
            dtau(2)=par(5)*xx(2,1);
        end
    end
end
%% Otherwise raise error
% Raise error if the requested derivative does not exist
if isempty(dtau)
    error(['SYS_DTAU, delay %d: requested derivative ',...
        'nx=%d, np=%d does not exist!'],ind, nx, np);
end
end
```

Listing 7: Jacobians of the delay function `sd_tau` in Listing 4 for sd-DDE tutorial example (11).

B. Octave compatibility considerations

`nargin` incompatibility The core DDE-BifTool code of version v2.0x was likely `octave` compatible. The changes to 3.1.1, replacing function names by function handles, broke this compatibility initially, because, for example, the call `nargin(sys_tau)` gives an error message in `octave` (version 3.2.3) if `sys_tau` is a function handle. To remedy this problem the additional field `tp_del` is attached to the structure `funcs` defining the problem. The field `funcs.tp_del` is set in `set_funcs` (see Section 7.1 and Table 3).

As of version 3.8.1, `octave` also gives an error message when one loads function handles as created using `set_funcs` from a data file. Function handles have to be re-created after a `clear` or a restart of the session. For an up-to-date list of known differences in syntax and semantics between matlab and `octave` see `http://www.gnu.org/software/octave`.

Output The gradual updating of plots using `drawnow` slows down for the `gnuplot`[3]-based plot interface of `octave` (as of version 3.2.3) as points get added to the plot. Setting the field `continuation.plot` to `0.5` (that is, less than 1 but larger than 0), prints the values on the screen instead of updating the plot. The `fltk`-based plotting interface of `octave` does not appear to experience this slow-down.

Useful options to be set in `octave`:

- `graphics_toolkit('fltk')` sets the graphics toolkit to the `fltk`-based interface (faster plotting);
- `page_output_immediately(true);` prints out the results of any `fprintf` or `disp` commands immediately;
- `page_screen_output(false);` stops paging the terminal output.

[3] `http://www.gnuplot.info/`