DDE-BIFTOOL v. 3.1 Manual — Bifurcation analysis of delay differential equations

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

December 16, 2024

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 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, S. G. Janssens, Y. A. Kuznetsov: Switching to nonhyperbolic cycles from codimension two bifurcations of equilibria of delay differential equations. Preprint https://arxiv.org/abs/1903.08276, 2017.

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:

```
BSD 2-Clause license

Copyright (c) |year|, K.U. Leuven, Department of Computer Science, K.
Engelborghs, T. Luzyanina, G. Samaey.  D. Roose, K. Verheyden, J. Sieber, B.
Wage, D. Pieroux

All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1.  Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

2.  Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
```

SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Download Upon acceptance of the above terms, one can obtain the package DDE-BIFTOOL (version 3.1) 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 3.2 to 4.0a

- Distributed delays and renewal equations: added experimental basic support for two types of distributed delays, approximated by many discrete delays with fixed relative (but not necessarily equidistant) spacing. Demos: `renewal_demo` at `../demos/renewal_demo/html/renewal_demo.html`, `daphnia_demo` at `../demos/renewal_demo/html/daphia_demo.html`
- Build-up of `funcs` structures: one may now create `funcs` structures with "more variables than equations" and provide a non-square left-hand side coefficient matrix M. Then one may create a structure for combining several `funcs` structures using `dde_add_funcs`, `dde_combined_funcs`, `dde_add_dist_delay`. See demos `renewal_demo` and `daphnia_demo`.
- Discrete symmetries and tracking or suppressing of symmetry breaking bifurcations at the linear algebra level: demo `pendulum` (`balancing_demo.m`) shows tracking of symmetry breaking bifurcations for $\mathbb{Z}_2$-symmetry.
- Continuous symmetries: support for rotational symmetries and translation symmetries is shown in demos `rotsym_demo`, `ddae_demo` and `renewal_demo` (`renewal_discrete_demo`).
- Added demos:
  - `pendulum` (`balancing_demo.m`, a system with reflection symmetry and symmetry-breaking bifurcations [51],
  - `renewal_demo`, a scalar renewal equations, where the integral over the past is approximated by an integral over a piecewise polynomial interpolant over many discrete delays [8];
  - `daphnia_demo`, a size structured population model with state-dependent implicitly given maturation age, the general problem class is discsussed in [14], the parameters of the demo come from [1]

2.2. Changes from 3.2a to 3.2b

- Added ability to perform bifurcation analysis on neutral DDEs via delay differential algebraic equations, by permitting a constant possibly singular matrix M in front of the left-hand side derivative $x'(t)$. A demo `ddae_demo` at `../demos/ddae_demo/html/ddae_demo.html` on a laser model with rotational symmetry is included. The default, when M is not provided by the user, is the identity matrix.

- **(internal)** The methods for solving nonlinear systems and the creation of defining systems have been separated. This enables the user to replace DDE-BIFTOOL's nonlinear methods with other standard methods, such as `fsolve` from Matlab's optimization toolbox, or the general continuation package COCO [11]. New function:

  ```
  [f,x0,data]=p_correc_setup(funcs,point,free_par,method,...)
  ```

  after which one may call, for example, `[x,success]=dde_nsolve(f,x0,method)` (the Newton iteration routine coming with DDE-BIFTOOL) or

  ```
  x=fsolve(f,x0,...
    optimoptions('fsolve','SpecifyObjectiveGradient',true))
  ```

  when using the optimization toolbox routine `fsolve`.

- A set of demos in folder `../demos/coco` and the functions in folder `../ddebiftool_coco` demonstrate how to create an interface to other nonlinear packages by creating an interface for COCO. The COCO atlas algorithm has additional features such as multi-parameter (e.g. surface) continuation and online monitoring of monitoring parameters that are currently not implemented in DDE-BIFTOOL's native algorithm (`br_contn`).

- Support for storing collocation polynomials on non-equidistant grids on each subinterval, together with pre-defined options of storing the polynomials on Chebyshev grids (options `'submesh'`, `'cheb'` for `SetupPsol`) for periodic orbits. This permits collocation polynomials of arbitrarily large degree d. Note that the computation of error estimates for remeshing computes the dth derivative, which may become affected by numerical overflow for very large degrees.

  The `method.point` field `collocation_parameters` can now have one of the names `'legendre'` or `'cheb'` to specify in which points on each collocation intervals the DDE is enforced.

- Additional demo `../demos/linear_stability` demonstrates problems that may occur when computing linear stability of equilibria.

- **(internal)** The routines for computation of folds of periodic orbits have been generalized such that they can be used for problems with rotational symmetry.

- **(internal, may lead to small changes in behaviour)** Utility functions `SetupStst`, `SetupHopf`, `SetupFold`, `SetupTorusBifurcation`, `SetupPOfold` etc now perform their initial corrections orthogonal to the tangent of the nullspace of the Jacobian in the initial guess. Correspondingly, their initial step is along the tangent, affecting the meaning of the initial stepsize. This should be more robust (permitting starting in folds) but may lead to small changes in the computational behaviour (number of steps required along branch etc).

- New utility function `Plot2dBranch` aids quick-and dirty plotting. Other plotting functions are still available.

2.3. Changes from 3.1.1 to 3.2a

- Added normal form analysis and utility functions for branch switching from some steady-state bifurcations of codimension 2 to periodic orbit bifurcation of codimension 1 (for state-dependent and constant delays). Demos:
  - `../demos/minimal_demo/html/minimal_demo.html`,
  - `../demos/minimal_demo/html/minimal_demo.html`,
  - `../demos/cusp/html/cusp.html`.

  (demos `minimal_demo`, `cusp` and tutorials.
- Added tutorials in folder `demos` (by M. M. Boscchaert).
- Added ability to automatically generate right-hand side and derivatives using symbolic toolbox (folder `ddebiftool_extra_symbolic`). Most demos use these, such as demos `minimal_demo`, `cusp`, `MackeyGlass`, `neuron`. If the symbolic toolbox for matlab is not available, the right-hand side and derivatives can also be generated with GNU Octave and its `sympy` package.
- Changed default stability computation to Chebyshev-based pseudo-spectral method, as in TRACE-DDE [7] (`dde_stst_eig_cheb`).
- Added experimental options `'sparse'` for computation of periodic orbits and their stability.

2.4. 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.

2.5. 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 [55].

2.6. 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 [54].
- **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] [35] or GNU Octave[2], both widely used environments for scientific computing. The aim of the package is to provide a tool for numerical 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). The equations may also be of *neutral* type, when they are specified in the form of delay differential algebraic equations (here called NDDEs). It also allows users to compute homoclinic and heteroclinic orbits in DDEs (not NDDEs) 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 (for NDDEs computations of roots closest to 0 is recommended);
- continuation of steady state folds and Hopf bifurcations (typically in two system parameters);
- continuation of periodic orbits using polynomial 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 (alternatively, the Floquet multipliers closest to some complex value c);
- 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`;
- branching off from codimension-two steady-state bifurcations to secondary codimension-one bifurcations using the extension `ddebiftool_extra_nmfm` and utility functions from `ddebiftool_utilities`;
- continuation of connecting orbits using the appropriate number of parameters (only for problems with constant discrete delays);
- approximate distributed delays and renewal equations by a large number of discrete delays;
- impose symmetries on solution components to defining equations to track or suppress symmetry-breaking bifurcations.

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

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

A practical difference to AUTO, MatCont or COCO 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. Earlier versions of the manual for earlier versions of DDE-BIFTOOL continue to be available at the web addresses

versions $\geqslant$ 3.0 `http://arxiv.org/abs/1406.7144`
versions $\leqslant$ 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 6.2, 7.4). 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 references [5, 17, 18, 31, 38] and the more recent references [3, 36, 32, 13, 37]. Several excellent books contain introductions to dynamical systems and bifurcation theory of ordinary differential equations, see, e.g., [2, 9, 29, 39, 47]. For background on numerical continuation methods one may refer to the textbooks [11, 15, 28].

Tutorial demos Demos such as `demo1`, `minimal_demo` 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.

For versions prior to 3.0 constructing a branch required creating two initial guesses and correcting them to obtain the first two points along the branch. From 3.0 onwards these recurring tasks have been wrapped in utility functions in folder `ddebiftool_utilities`. These utility functions provide additional robustness by correcting and predicting along the tangent and ensuring default initialization of all fields in stuctures. For these reasons using the utility functions is now the preferred way of using DDE-BIFTOOL. Most demos rely on them. Some demos (see `demo1`) still demonstrate the use of low-level functions.

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 [16, 15],
`MatCont`	url: `http://sourceforge.net/projects/matcont/`	for MATLAB [12, 28], and
COCO	url: `http://sourceforge.net/projects/cocotools`	for MATLAB [11].

For delay differential equations the package

`knut`	url: `https://rs1909.github.io/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 [53, 45]. 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 [43, 10, 25, 24, 48, 30]. Julia has a suite of time integrators able to treat state-dependent and constant delays [44]. Of these, only XPPAUT has a graphical interface (and allows limited stability analysis of steady state solutions of DDEs along the lines of [41]). TRACE-DDE is a MATLAB tool (with graphical interface) for linear stability analysis of linear constant-coefficient DDEs [7].

4. Minimal example – Mackey-Glass equation

A well-known example for the effects of delayed nonlinear feedback is the Mackey Glass equation [27, 26],

$$x'(t) = b\frac{x(t-\tau)}{1+x(t-\tau)^n} - gx(t) \tag{1}$$

with $x(t) \geqslant 0$ and parameters b, n, delay τ, g. The non-trivial equilibria $x_{eq} = \sqrt[n]{b/g-1}$ are known to become unstable in a Hopf bifurcation. The periodic orbits emerging then undergo period doubling cascades. The demo `MackeyGlass_minimal` demonstrates the minimal information the user has to provide to obtain bifurcation diagrams.

DDE-BIFTOOL consists of a set of functions that need to get loaded into the user's path. The user will have to modify the definition of `base` to point to the correct folder. Some additional folders contain functions providing additional functionality, such as normal form computations and symbolic derivative generation.

```
%% load DDE-Biftool into path
clear
base=[pwd(),'/../../']; % set to folder where DDE-Biftool is
addpath('../../ddebiftool',...       % load base routines
    '../../ddebiftool_extra_psol',...% for  periodic orbit bifurcations
    '../../ddebiftool_utilities');   % interface routines
format compact
```

We define the right-hand side initially in an arbitrary format (with 5 arguments $x(t)$, $x(t-\tau)$, b, n, g).

```
[ib,in,itau,ig]=deal(1,2,3,4);        % define parameter indices
f=@(x,xd,b,n,g)b.*xd./(1+xd.^n)-g.*x; % r.h.s., all vectorized for speed-up
```

Then we convert the right-hand side to the general format for a DDE-BIFTOOL right-hand side, $f(x,p)$, where the state varaiables are passed on as $x \in \mathbb{R}^{n_x\times(n_\tau+1)}$, and the parameter p is passed on as $\mathbb{R}^{1\times n_p}$. In our case $n_x = 1$, $n_\tau = 1$, $n_p = 4$.

```
sys_rhs=@(xx,p)f(xx(1,1,:),xx(1,2,:),p(1,ib,:),p(1,in,:),p(1,ig,:));
```

We have *vectorized* the right-hand side, that is, it is in a form that permits it to be called with many different parameters and states simultaneously, so with $x \in \mathbb{R}^{n_x\times(n_\tau+1)\times n_{vec}}$, $p \in \mathbb{R}^{1\times n_p\times n_{vec}}$ with arbitrary n_{vec}, to result in n_{vec} values of `sys_rhs`. Vectorization is optional but strongly recommended if periodic-orbit computations are required.

The call to `set_funcs` now takes the right-hand side, the user information that the delay is constant and equals parameter number `itau`, and that the right-hand side is vectorized.

```
funcs=set_funcs('sys_rhs',sys_rhs,'sys_tau',@()itau,...
     'x_vectorized',true,'p_vectorized',true);
```

The structure `funcs` contains additional information generated by `set_funcs` (e.g. derivatives obtained by finite differences). The final piece of information the user needs to provide are initial guesses for the equilibrium.

```
[b,n,tau,g]=deal(2, 10, 0,1);   % initial parameters
x0=(b/g-1)^(1/n);               % initial non-trivial equilibrium
```

Start of bifurcation analysis Bifurcation analysis starts by initializing a solution branch, here a branch of equilibria.

```
bounds={'max_bound',[itau,2;ib,5],'max_step',[0,0.3]};
[eqbr,suc]=SetupStst(funcs,'x',x0,'parameter',[b,n,tau,g],...
    'step',0.1,'contpar',itau,bounds{:})
```

The call to `SetupStst` contains `funcs`, initial state guess after `'x'`, initial parameter after `'parameter'` and the index of the parameter we intend to vary (τ), a maximal step along the branch during continuation after `'max_step'`, and a boundary for the two parameters we intend to use as bifurcation parameters, τ and b. The call creates a *solution branch* `eqbr` with initially 2 points, which are stored in a field called `'point'`. They can be checked:

```
eqbr =   struct with fields:
       method: [1x1 struct]
    parameter: [1x1 struct]
        point: [1x2 struct]
suc =    1
```

```
>> eqbr.point(2)
ans =   struct with fields:
         kind: 'stst'
    parameter: [2 10 0.1 1]
            x: 1
    stability: []
         nmfm: []
         nvec: []
         flag: ''
```

Some information about the points such as stability has not yet been computed, leaving some fields empty. Next we continue the branch (in parameter τ as specified in `SetupStst`). The continuation "live-plots" if requested. The live plot is showing a straight line since the delay does not affect the location of the equilibrium.

```
figure(1);clf;ax1=gca;
eqbr=br_contn(funcs,eqbr,10,'ax',ax1); % continue, stop after max 10 steps
```

However, stability along the branch changes, as the call to `br_stabl` finds:

```
[eqbr,nunst_eqs]=br_stabl(funcs,eqbr);
ihopf=find(diff(nunst_eqs));
fprintf('Hopf bifurcation near point %d, tau=%g\n',...
   ihopf,eqbr.point(ihopf).parameter(itau));
```

```
parameter bound 2 for parameter(3) reached
br_contn warning: boundary 1 hit: parameterbounds.
Hopf bifurcation near point 4, tau=0.364
>> nunst_eqs.'
ans =     0     0     0     0     2     2     2     2     2     2     2
>> eqbr.point(5).stability.l0
ans =      0.14702 +      3.514i      0.14702 -      3.514i
```

The second output of `br_stabl` lists the number of unstable eigenvalues, the first output is our branch where now all points contain stability information. For example, point 5 has already an unstable pair of complex eigenvalues. We now have a choice, we may either track the Hopf bifurcation in two parameters, or branch off, following the periodic orbits. Let us follow the Hopf bifurcation first:

```
[hopfbr,suc]=SetupHopf(funcs,eqbr,ihopf,...
    'contpar',[ib,itau],'dir',ib,'step',1e-1,bounds{:})
```

The initialization with `SetupHopf` passes on the problem `funcs`, the branch and point number that was close to the Hopf bifurcation on the equilibrium branch `eqbr`, the indices of the two continuation parameters (`[ib,itau]`), the initial direction and the initial step. The resulting branch of Hopf bifurcation points contains points with additional information such as the Hopf frequency ω:

```
>> hopfbr.point(1)
ans =   struct with fields:
         kind: 'hopf'
    parameter: [5 10 0.24742 1]
            x: 1.1487
            v: -1
        omega: 6.9282
...
```

The continuation that follows live plots the curve as computed in the two-parameter plane (τ, b):

```
figure(2);clf;ax2=gca; xlabel('b');ylabel('tau');
hopfbr=br_contn(funcs,hopfbr,30,'ax',ax2);
hopfbr=br_rvers(hopfbr);
hopfbr=br_contn(funcs,hopfbr,30,'ax',ax2);
[hopfbr,nunst_h]=br_stabl(funcs,hopfbr);
```

The stability, as determined by `br_stabl`, finds that all Hopf points are transversally stable:

```
>> nunst_h.'
ans =  Columns 1 through 13
     0     0     0     0     0     0     0     0     0     0     0     0     0
  Columns 14 through 26
     0     0     0     0     0     0     0     0     0     0     0     0     0
  Columns 27 through 32
     0     0     0     0     0     0
```

Next we branch off at the Hopf point first detected and follow periodic orbits.

```
[per_orb,suc]=SetupPsol(funcs,eqbr,ihopf,'intervals',20,'degree',4,...
    'max_step',[itau,0.5])
```

The initialization does not require a parameter if the same parameter is used as in the equilibrium branch (`itau`). We also specify two *discretization quantities* typical for periodic orbits. A periodic orbit is approximated by a piecewise polynomial consisting of n_{int} pieces of degree n_{deg}, in this case, $n_{\mathrm{deg}} = 4$ and $n_{\mathrm{int}} = 20$. Higher numbers result in higher accuracy but more computational expense. DDE-BIFTOOL does not issue a warning about inaccurate approximation. The number of variables for the nonlinear system to be solved is in this case $n_x(n_{\mathrm{int}}n_{\mathrm{deg}} + 1) + 2$. In this case this equals 83, of which 81 are for storing x on 81 mesh points, while 2 more variables are for the unknown period and the parameter τ to be varied. The initialization produces a branch with

two points. The point structure in the `'point'` field of `per_orb` contains all needed information to plot an orbit:

```
per_orb =   struct with fields:
       method: [1x1 struct]
    parameter: [1x1 struct]
        point: [1x2 struct]
suc =   1
>> pt=per_orb.point(2)
pt =   struct with fields:
         kind: 'psol'
    parameter: [2 10 0.47129 1]
         mesh: [0 0.010983 0.021967 0.03295 0.043934 0.055096 0.066257 ... ]
       degree: 4
      profile: [0.99998 0.99929 0.9986 0.99792 0.99725 0.99658 0.99593 ... ]
       period: 1.6238
...
>> figure(4);clf;plot(pt.mesh,pt.profile,'o-');
>> xlabel('time/period');ylabel('x(t)');
```

The continuation produces a branch of orbits with increasingly complicated time profiles, but the branch loses stability at some parameter:

```
figure(1);
per_orb=br_contn(funcs,per_orb,60,'ax',ax1);
[per_orb,nunst_per,dom_per]=br_stabl(funcs,per_orb);
```

This is detected by the second output of `br_stabl`, `nunst_per`. At the point (number `ipd=23`) where the stability changes, we inspect the Floquet multipliers and find that the dominant non-trivial Floquet muliplier is real and less than -1, indicating that the branch has encountered a period doubling.

```
>> nunst_per.'
ans =  Columns 1 through 13
     0     0     0     0     0     0     0     0     0     0     0     0     0
  Columns 14 through 26
     0     0     0     0     0     0     0     0     0     0     1     1     1
  Columns 27 through 39
     1     1     2     2     2     3     3     3     3     4     4     4     5
...
>> ipd=find(diff(nunst_per)==1,1,'first')
ipd =  23
>> per_orb.point(ipd+1).stability.mu(1:3)
ans = -1.1444
            1
      0.11868
>> dom_per(ipd+1)
ans = -1.1444
```

Continuing the period doubling in two parameters follows the pattern of continuing a Hopf bifurcation. However, the initialization `SetupPeriodDoubling` modifies the problem, such that it also returns a new `funcs` stucture for the *extended problem*, here called `pdfuncs`.

```
[pdfuncs,pdbr1,suc]=SetupPeriodDoubling(funcs,per_orb,ipd,...
    'contpar',[ib,itau],'dir',ib,'step',1e-1,bounds{:})
```

The initial points now have dimension 3 in field `'profile'`. The extra dimensions contain the real and imaginary part of the periodized Floquet eigenfunction for multiplier -1:

```
>> pdbr1.point(1)
ans =   struct with fields:
          kind: 'psol'
     parameter: [2.1501 10 1.2969 1 1 3.8156]
          mesh: [0 0.018622 0.037245 0.055867 0.074489 0.09322 0.11195 ... ]
        degree: 4
       profile: [3x81 double]
        period: 3.8156
...
```

After initialization continuation works as usual but we pass on the extended problem structure `pdfuncs`.

```
figure(2);
pdbr1=br_contn(pdfuncs,pdbr1,30,'ax',ax2);
pdbr1=br_rvers(pdbr1);
pdbr1=br_contn(pdfuncs,pdbr1,30,'ax',ax2);
[pdbr1,nunst_pd]=br_stabl(pdfuncs,pdbr1);
```

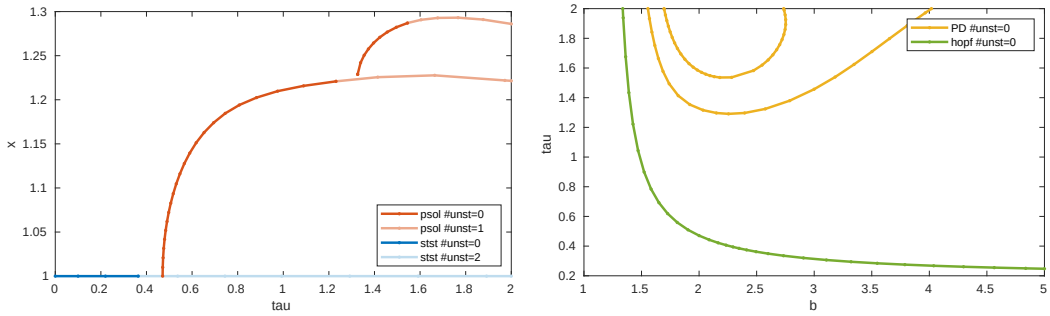
In particular, `br_stabl` is aware of two critical Floquet multipliers and reports only the non-trivial ones, which are all stable here:

```
>> nunst_pd.'
ans =  Columns 1 through 13
     0     0     0     0     0     0     0     0     0     0     0     0     0
  Columns 14 through 23
     0     0     0     0     0     0     0     0     0     0
```

We now branch off at the period doubling, detect a secondary period doubling, and then track this secondary period doubling in two parameters repeating the steps above.

```
[per2,suc]=DoublePsol(funcs,per_orb,ipd);
per2=br_contn(funcs,per2,60,'ax',ax1);
[per2,nunst_p2,dom_p2,triv_defect2]=br_stabl(funcs,per2);
ipd2=find(diff(nunst_p2)==1,1,'first');
[pd2funcs,pdbr2,suc]=SetupPeriodDoubling(funcs,per2,ipd2,...
    'contpar',[ib,itau],'dir',ib,'step',1e-1,bounds{:});
figure(2);
pdbr2=br_contn(pd2funcs,pdbr2,30,'ax',ax2);
pdbr2=br_rvers(pdbr2);
pdbr2=br_contn(pd2funcs,pdbr2,30,'ax',ax2);
[pdbr2,nunst_pd2]=br_stabl(pd2funcs,pdbr2);
```

The parameters of each point `i` in each branch *br* are in the field *br*`.point(i).parameter`, the state variables are in *br*`.point(i).x` (for steady states, Hopf bifurcations and fold bifurcations), and in *br*`.point(i).profile` and *br*`.point(i).period` (for periodic orbits and their bifurcations). Eigenvalues (after computation with `br_stabl`) are in *br*`.point(i).stablity.l0` for equilibria and in *br*`.point(i).stablity.mu` for periodic orbits. From this information bifurcation diagrams can be constructed. The auxiliary function `Plot2dBranch` provides a first overview:



(a) One-parameter diagram in τ. (b) Two-parameter diagram in (b, τ)-plane.

Figure 1: Resulting first bifurcation diagrams

```
figure(5);clf;
Plot2dBranch({eqbr,per_orb,per2}); % 1d diagram
legend('Location','southeast')
xlabel('tau');ylabel('x');
figure(3);clf;hold on;xlabel('b');ylabel('tau');
Plot2dBranch(hopfbr);              % 2d diagram
Plot2dBranch({pdbr1,pdbr2},'funcs',pd2funcs);
```

The stability boundaries are only accurate up to stepsize along the branch in this plot. Refinement routines such as `LocateSpecialPoints` and `MonitorChange` are demonstrated in other demos.

5. Structure of DDE-BIFTOOL

The structure of the package is depicted in figure 2. It consists of four layers.

Definition of DDE Layer 0 contains the system definition and consists of routines which allow other routines 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 7. 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. It is strongly recommended that this structure is created using constructors such as `set_funcs`, `set_symfuncs`, `SetupTorusBifurcation`, etc do ensure that all fields are consistently initialized. ***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 defining systems and numerical methods used are explained briefly in section 11, more details can be found in the papers [41, 22, 21, 20, 23, 40, 46] and in [19]. Its functionality is hidden by and used through layers 2 and 3. Many of these functions have the prefix `dde_`.

Layer 2 — Point structures Solutions are referred to as *points* in DDE-BIFTOOL. These are Matlab `struct`'s for which DDE-BIFTOOL has created abstract functions that can create or manipulate them. A user (or creator of extensions) can in principle create new *kinds* of points, for which a specific implementation of point methods and residuals for layer 1 may have to be then provided.

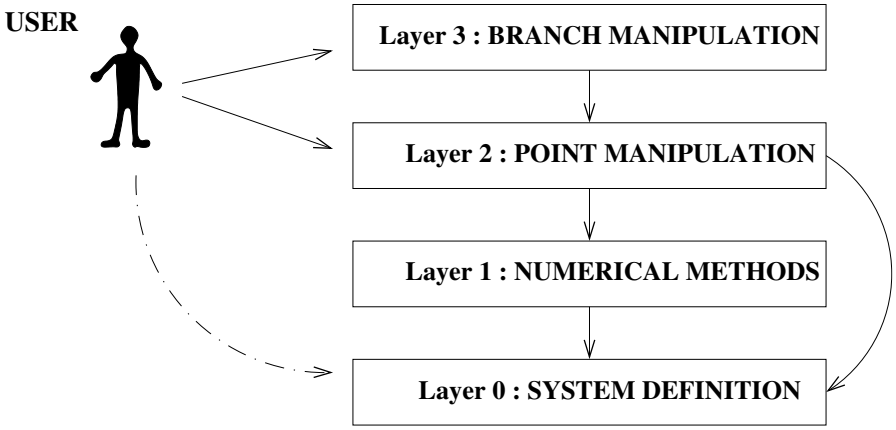


Figure 2: The structure of DDE-BIFTOOL. Arrows indicate the calling (—) or writing (·—) of routines in a certain layer.

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

1. a steady state point (abbreviated `'stst'`),
2. steady state Hopf (abbreviated `'hopf'`) or
3. fold (abbreviated `'fold'`) bifurcation point,
4. a periodic solution point (abbreviated `'psol'`) or
5. a connecting orbit point (abbreviated `'hcli'`).

Furthermore, a point can contain additional information concerning its stability, normal form information or flags. Routines are provided to compute individual points, to compute and plot their stability and to convert points from one type to another.

For each point type there exists a simple constructor with the name `['dde_',kind,'_create']`, where `kind` equals the entry of the `'kind'` field in the point structure. These constructors ensure that all fields are always present and in the right order.

Layer 3 — Branch structures Layer 3 contains routines to manipulate branches. Names of routines in this layer start with "`br_`". A branch is a 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 8, 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 9 respectively 10.

Constructor and wrapping utilities During construction of branches earlier versions of DDE-BIFTOOL required a combination of calls to routines in layer 2 and 3. Constructors in folder `../ddebiftool_utilties` automate these tasks (e.g., `SetupStst`). They also provide additional automated extraction and creation of stability information (`GetStability`, `MonitorChange`), plotting (`Plot2dBranch`), branching (e.g. `ChangeBranchParameters`, `SetupPsol`, `BranchFromCodimension2`) and normal form processing (e.g. `LocateSpecialPoints`).

The distinction between layers (esp. layer 2 and 3) and utilities and extensions is for historical reasons only. For example, for computation of linear stability along a branch the utility function `GetStability` and the layer 3 function `br_stabl` both call the layer 2 function `p_stabil`, which in turn will call, for example, the layer 1 function `dde_stst_eig` if the points in the branch have a suitable `'kind'` field. The functions `br_stabl` and `GetStability` have additional features such as the counting of unstable eigenvalues depending on `'kind'` field and the rules provided in `pointtype_list()`.

6. Delay differential equations

This section introduces the mathematical notation that we refer to in this manual to describe the problems solved by DDE-BIFTOOL. Internally, all code except compuation of connecting orbits treats DDEs with constant delays and with state-depndent delays in a uniform interface.

6.1. Equations with constant delays

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

$$M\frac{d}{dt}x(t) = f(x(t), x(t-\tau_1), \ldots, x(t-\tau_{n_\tau}), p), \tag{2}$$

where M is a constant $n_f \times n_x$ matrix, $x(t) \in \mathbb{R}^{n_x}$, $f : \mathbb{R}^{n_x(n_\tau+1)} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_f}$ is a nonlinear smooth function depending on a number of parameters $p \in \mathbb{R}^{n_p}$, and delays $\tau_i > 0$, $i = 1, \ldots, n_\tau$. Call τ the maximal delay,

$$\tau = \max_{i=1,\ldots,n_\tau} \tau_i.$$

The linearization of (2) around a solution $x^*(t)$ is the *variational equation*, given by,

$$M\frac{d}{dt}y(t) = \sum_{i=0}^{n_\tau} A_i(t)y(t-\tau_i), \tag{3}$$

where $\tau_0 = 0$ and, using $f \equiv f(x^0, x^1, \ldots, x^{n_\tau}, p)$,

$$A_i(t) = \frac{\partial f}{\partial x^i}(x^*(t-\tau_0), \ldots, x^*(t-\tau_{n_\tau}), p),\ i = 0, \ldots, n_\tau. \tag{4}$$

For $n_f = n_x$ and regular M eq. (2) is a classical DDE. For singular M, equations of type (2) can create a variety of different types of equation, including differential algebraic equations, backward-forward equations and neutral DDEs (NDDEs). DDE-BIFTOOL is tested only for neutral DDEs, DDEs with implicitly given delay, and index-1 delay-differential-algebraic equations (DDAEs), where additional equations approximating distributed delays have been added.

Neutral DDEs as DDAEs Consider a NDDE of the form analysed by Hale& Verduyn-Lunel [32]

$$\frac{d}{dt}\left[z(t) + f_\ell(z(t), z(t-\tau_1), \ldots, z(t-\tau_{n_\tau}), p)\right] = f_r(z(t), z(t-\tau_1), \ldots, z(t-\tau_{n_\tau}), p) \tag{5}$$

with n_z-dimensional z. Equation (5) can then be put into the form (2) of dimension $n_x = 2n_z$ with

$$M = \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix}, \quad x(t) = \begin{bmatrix} y(t) \\ z(t) \end{bmatrix}, \quad f\left(\begin{bmatrix} y_0 \\ z_0 \end{bmatrix}, \begin{bmatrix} y_1 \\ z_1 \end{bmatrix}, \ldots \begin{bmatrix} y_{n_\tau} \\ z_{n_\tau} \end{bmatrix}, p\right) = \begin{bmatrix} f_r(z_0, z_1, \ldots z_{n_\tau}, p) \\ z_0 + f_\ell(z_0, z_1, \ldots z_{n_\tau}, p) - y_0 \end{bmatrix}$$

and the same delays $\tau_1, \ldots, \tau_{n_\tau}$.

Distributed delays use the interface for state-dependent delays, so see Section 6.2 for their treatment.

6.1.1. Steady states

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

$$x^*(t) \equiv x^* \in \mathbb{R}^{n_x}, \text{ with } f(x^*, x^*, \ldots, x^*, p) = 0,$$

then the matrices $A_i(t)$ are constant, $A_i(t) \equiv A_i$, and the corresponding variational equation (3) leads to a *characteristic equation*. Define the $n \times n$-dimensional matrix Δ as

$$\Delta(\lambda) = \lambda M - \sum_{i=1}^{n_\tau} A_i e^{-\lambda \tau_i}. \tag{6}$$

Then the characteristic equation reads,

$$\det(\Delta(\lambda)) = 0. \tag{7}$$

Equation (7) 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 (7) have negative real part; it is unstable if there exists a root with positive real part. For regular M (classical DDEs) 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.

6.1.2. 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 (3) 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, for classical DDEs ($M = I$) the operator $S(T, 0)^k = S(kT, 0)$ 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 exponentially unstable if there exists a multiplier with modulus larger than unity. *Local bifurcations* are characterized by additional Floquet multipliers on the unit circle:

- Fold of periodic orbits (`POfold`): double Floquet multiplier at 1;
- Period doubling (`PeriodDoubling`): Floquet multiplier at -1;
- Torus vbifurcation (`TorusBifurcation`): complex pair of Floquet multipliers $\exp(\pm 2i\pi\omega)$ with $\omega \in (0, 1/2)$.

6.1.3. Connecting orbits

We call a solution $x^*(t)$ of eq. (2) at $p = p^*$ a *connecting orbit* if the limits

$$\lim_{t\to-\infty} x^*(t) = x^-, \qquad \lim_{t\to+\infty} x^*(t) = x^+, \tag{8}$$

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

6.2. Equations with state-dependent delays

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

$$\begin{cases} M\dfrac{d}{dt}x(t) = f(x_0, x_1, \ldots, x_{n_\tau}, p), \quad \text{with} \\ \qquad x_j = x(t - \tau_j(x_0, \ldots, x_{n_{\tau,j}}, p) \quad (\tau_0 = 0, j = 1, \ldots, n_\tau), \end{cases} \tag{9}$$

where $x(t) \in \mathbb{R}^{n_x}$, $n_{\tau,j} < j$, and

$$\begin{aligned} f &: \mathbb{R}^{n_x \times (n_\tau+1)} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_f} \\ \tau_j &: \mathbb{R}^{n_x \times n_{\tau,j}} \times \mathbb{R}^{n_p} \to [0, \infty) \end{aligned}$$

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

Linearization The linearization around a solution $(x^*(t), p^*)$ of (9) (the *variational equation*) with respect to x is given by (see [33], 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 $j \geqslant 1$)

$$\begin{aligned} M\frac{d}{dt}y(t) &= \sum_{j=0}^{n_\tau} A_j(t) Y_k \\ Y_0 &= y(t) \\ Y_j &= y(t - \tau_j^*(t)) - (x^*)'(t - \tau_j(t)) \sum_{k=0}^{n_{\tau,j}} B_{j,k}(t) Y_k, \quad (j = 1 \ldots n_\tau) \end{aligned} \tag{10}$$

where $(x^*)'(t) = dx^*(t)/dt$, and

$$\begin{aligned} A_j(t) &= \frac{\partial f}{\partial x^j}(x_0^*, x_1^*, \ldots, x_{n_\tau}^*, p^*) \in \mathbb{R}^{n\times n}, \quad (j = 0, \ldots, m), \\ B_{j,k}(t) &= \frac{\partial \tau_j}{\partial x^k}(x_0^*, x_1^*, \ldots, x_{n_{\tau,j}}^*, p^*) \in \mathbb{R}^{1\times n}, \quad (k = 0, \ldots, j-1, j = 1, \ldots, n_\tau). \end{aligned} \tag{11}$$

If $(x^*(t))$ corresponds to a steady state solution, then $x^*(t) = x_0^* = \ldots = x_{n_\tau}^* \equiv x^* \in \mathbb{R}^{n_x}$, and $\tau_j^*(t) \equiv \tau_j(x^*, \ldots, x^*, p^*)$ for all $j \geqslant 1$, with

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

Then the matrices $A_i(t)$ are constant, $A_i(t) \equiv A_i$, and the $n_x \times n_x$-matrices $(x^*)'(t-\tau_j(t))B_{j,k}(t)$ consist of zero elements only. In this case, the corresponding variational equation (10) is a constant delay differential equation and it leads to the characteristic equation (7), i.e. a characteristic equation with constant delays. Hence the stability analysis of a steady state solution of (9) is identical to the stability analysis of (2).

Note that the right-hand side f, when considered as a functional mapping a history segment into $\mathbb{R}^{n_f}$ is not locally Lipschitz continuous. This creates technical difficulties when considering a sd-DDE of type (9) as an infinite-dimensional system, because the solution does not depend smoothly on the initial condition (see Hartung *et al.* [33] for a detailed review). However, periodic boundary-value problems for (9) can be reduced to finite-dimensional systems of algebraic equations that are as smooth as the coefficient functions f and τ_j [50]. 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 [50]).

Moreover, Mallet-Paret and Nussbaum [42] proved that the stability of the linear variational equation (10) indeed reflects the *local stability* of the solution $(x^*(t))$ of eq. (9). For details on the relevant theory and numerical bifurcation analysis of differential equations with state-dependent delay see [40, 33, 52] and the references therein.

6.3. Equations with distributed delays

Experimentally DDE-BIFTOOL permits provision of an initially non-square DDAE problem

$$M\dot{x}(t) = f(x(t), x(t-\tau_1), \ldots, x(t-\tau_{n_\tau}), p), \tag{12}$$

where $n_x > n_f$. To this problem one can then append further equations. This mechanism is specifically intended to add equations defining distributed delays. Using `dde_add_dist_delay`, or the combination of `dde_create/add/close_chain_delay` one may add distributed delays, which are then entering (12) as the additional $n_x - n_f$ state components x_d. A problem with a distributed delay over $[t-\tau_d, t]$ is reduced to a problem with many discrete delays by one of the two following forms.

Simple integral over past A fixed mesh (s_μ) where $\mu = 1, \ldots, N_\mu$ ($N_\mu = n_{int} n_{deg}$) approximates functions of selected components x_i of the state over $[t-\tau_d, t]$ by a piecewise polynomial with n_{nint} pieces of degree n_{deg}. This introduces the algebraic equation

$$0 = \tau_d \sum_{\mu=1}^{N_\mu} w_\mu g(s_\mu \tau_d, x_i(t - s_\mu \tau_d), p_\ell) - x_d(t), \tag{13}$$

with quadrature weights w_μ, approximating

$$0 = \int_0^{\tau_d} g(s, x_i(t-s), p_\ell)ds - x_d(t). \tag{14}$$

The integral bound τ_d may be either a state $x_{i_{\tau,d}}(t)$ or a parameter $p_{i_{\tau,d}}$. Equation (13) introduces N_μ delays $\tau_\mu = s_\mu \tau_d$. Since the DDE-BIFTOOL interface for constant delays would require introducing and coupling the N_μ delays as parameters, distributed delays will be treated internally with the interface for state-dependent delays. The user has to specify the index $i_{\tau,d}$ for the bound, the index sets ℓ, i for the state components and parameters entering g, and the index (or index set) for $x_d(t)$, which is the variable representing the distributed delay).

One can in principle add multiple distributed delays of this type. The index sets for x_d and i can also overlap, thus, permitting nested/multiple integrals (both features not tested). A simple demo, `renewal_demo`, is described in Section 7.7.1.

Chain of nested integrals evaluated at selected times in the past A common scenario is that effects accumulated over the past at different stages enter the DDE, such that one adds a sequence of variables $y_{id,j}(s,t)$ and $y_{int,j}(s,t)$ for $j \leqslant n_c$ satisfying

$$\text{for } s \in [0, \tau_{max}] \quad y_{id,0}(s,t) = x_i(t-s) \quad \text{for a user-given index set } i, \tag{15}$$

$$\text{for } s \in [0, \tau_{max}] \quad y_{id,j}(s,t) = g_j(s, y_j(s,t), p_j) \quad \text{for } j > 0, \tag{16}$$

$$\text{for } s \in [0, \tau_{max}] \quad y_{int,j}(s,t) = y_{0,j} + \int_0^s y_{id,j}(s,t)ds \quad \text{for user-given } y_{0,j}. \tag{17}$$

For each argument $y_j(s,t)$ of g_j the user may select components from $y_{id,\nu}$ or $y_{int,\nu}$ with $\nu < j$ (so, for $j = 1$, $y_j(s,t) = x_i(t-s)$ or a subset of the components). Similarly, the parameters p_j and initial values $y_{0,j}$ are selected as index sets from ther parameter vector p and the state components x in (12). From this chain, one can then extract the distributed delays x_d as

$$0 = \sum_{k=1}^{n_d} S_k y_s(r_{s,k}\tau_{max}, t) - x_d(t). \tag{18}$$

In (18), the factors y_s are n_s user-chosen components from the chain $(y_{id,j}, y_{int,j})$. The $n_d \times n_s$ matrices S_k are user-given constants, while the upper bound of the τ_{max} of the chain and the fractions $r_{s,k}$ are states or parameters (determined by user-selected index sets). The chain variables $(y_{id,j}, y_{int,j})$ are represented on a fixed mesh (s_μ) on $[0, \tau_{max}]$ where $\mu = 1, \ldots, N_\mu$ ($N_\mu = n_{int} n_{deg}$), approximates functions of selected components x_i of the state over $[t-\tau_d, t]$ by a piecewise polynomial with n_{nint} pieces of degree n_{deg}. Thus, there will be N_μ delays $\tau_\mu = s_\mu \tau_{max}$ introduced by the chain. The quantities $r_{s,k}\tau_{max}$ will not be explicit delays. The values of $y_s(r_{s,k}\tau_{max}, t)$ will be obtained by polynomial interpolation from the values at $s_\mu \tau_{max}$.

7. System definition

This section explains several different ways in which the user can define their system of equations and the delays. We recommend that the user tries the various ways in the following order:

1. For initial exploration of equilibria, periodic orbits and stability analysis, it is likely sufficient for the user to define only the right-hand sides, as descibed in Sections 7.3.1 and 7.3.2 for constant (parameter) delays and Sections 7.4.1 and 7.4.2 for state-dependent delays.

Then they may use `set_funcs` as described in Section 7.6 to create the structure that is passed on to DDE-BIFTOOL routines. Partial derivatives will be approximated using finite diffferences. For speed-up of periodic orbit computations, a vectorized right-hand side should be provided (see, e.g., Listing 1). Distributed delays may be added in a second step after `set_funcs`, as described in Section 7.7.

2. Once basic analysis is successful and more advanced tasks such as normal form analysis or continuation of bifurcations are planned, partial derivatives should be provided. The easiest way to do this is using the symbolic toolbox and `dde_sym2funcs` (see Listing 3 in Section 7.3.3 for a constant delay example and Listing 7 in Section 7.4.3 for a state-dependent delay example). Especially normal form analysis requires higher-order derivatives of order 3 or greater, for which finite-difference approximations are unreliable.

3. If problems or inexplicable discrepancies between results using approaches in point 1 and in point 2 occur, the automatically generated expressions from the symbolic toolbox may be hard to debug. In this case, the user may provide derivatives manually. This is done in the form of directional derivatives, as shown in Listing 4 in Section 7.3.4 for constant delays (see also Section 7.4.4 for state-dependent delays).

4. For backward compatibility the user may also provide first- and in the form compatible with versions v. 3.1 and earlier, explained in Section 7.3.5 for constant delays and Section 7.4.5 for state-dependent delays. This is not recommended and will be wrapped and converted to directional derivatives internally up to first order. The second derivatives are no longer directly used. Only the `hcli` code uses the `sys_deri` format.

7.1. Change from DDE-BIFTOOL v. 3.1 to v. 3.2: symbolic generation

The folder `../ddebiftool_extra_symbolic` contains routines that enable the user to create right-hand sides, delays (if state-dependent), or integrands for distributed delays involving integrals and their derivatives through automatic code generation. The routines rely on either the symbolic toolbox of Matlab or the package `symbolic` of Octave (which relies on the `sympy` python library). In particular, right-hand sides and derivatives generated with the symbolic toolbox do not require the symbolic toolbox to be used in computations. Right-hand sides and derivatives generated with Octave can be used in Matlab (and vice versa).

Previous versions had external Mathematica scripts such as `../external_tools/genr_sys.mth`, provided by D. Pieroux (still available).

For manually implemented derivatives, the user may also provide directional derivatives instead of the more complex Jacobians.

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

Note that from DDE-BIFTOOL v. 3.0 all user-provided functions can be arbitrary function handles, collected into a structure using the function `set_funcs`, explained in section 7.6. 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 versions < 3.0. See the tutorials in `../demos/index.html` for examples of usage, and the function description in section 7.6 for details.

The new version also supports ***vectorized*** right-hand sides and delays, leading to speed-up in computations involving periodic orbits.

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

Listing 1: Definition for right-hand side of (19) as a variable. See Listing 2 for the vectorized version.

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

Listing 2: Alternative definition of the right-hand side of (19), vectorized in `x` and `p` for speed-up of periodic orbit and bifurcation computations.

7.3. Equations with constant delays

As an illustrative example we will use the following system of delay differential equations, taken from [49],

$$\begin{cases} \dot{x}_1(t) = -\kappa x_1(t) + \beta \tanh(x_1(t-\tau_s)) + a_{12} \tanh(x_2(t-\tau_2)) \\ \dot{x}_2(t) = -\kappa x_2(t) + \beta \tanh(x_2(t-\tau_s)) + a_{21} \tanh(x_1(t-\tau_1)). \end{cases} \tag{19}$$

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 (19) step by step to demonstrate the working pattern for DDE-BIFTOOL.

To define a system, the user should provide a set of MATLAB functions specifying the right-hand side and delays for system (19), and optionally their partial derivatives. These functions will then be collected into a strcture using the routine `set_funcs`, described in Section 7.6. The following paragraphs describe different methods how to provide or automatically generate these functions.

7.3.1. Direct provision of right-hand side — `sys_rhs`

The right-hand side is a function of two arguments. For our example (19), this would have the form (giving the right-hand side the name `neuron_sys_rhs`) given in Listing 1. Meaning of the arguments of the right-hand side function:

- `x` $\in \mathbb{R}^{n\times(n_\tau+1)}$ 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 ($i = 1\ldots, n_\tau$) are considered to be part of the parameters ($\tau_i = \eta_{j(i)}, i = 1, \ldots, n_\tau$). 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.

Vectorization An alternative (vectorized) form would be as given in Listing 2. Note the additional colon in argument `x` and in `p`, and compare to Listing 1. The form shown in Listing 2 can be called in many points and parameters along a mesh simultaneously, speeding up the computations during analysis of periodic orbits and Jacobians (for finite difference approximations).

```
ntau=3; % Set number of delays and parameter names
parnames={'kappa','beta','a12','a21','tau1','tau2','taus'};
%% Define system using symbolic algebra using arbitrary names
x=sym('x',[2,ntau+1]);
syms(parnames{:});
par=sym(parnames);
f=[-kappa*x(1,1)+beta*tanh(x(1,4))+a12*tanh(x(2,3));...
   -kappa*x(2,1)+beta*tanh(x(2,4))+a21*tanh(x(1,2))];
%% Differentiate and generate code, exporting it to sym_neuron
dde_sym2funcs(f,x,par,'filename','sym_neuron');
```

Listing 3: Definition for right-hand side and all derivatives of (19) using the symbolic toolbox. See Listing 1 for manually provided right-hand sides. See Section 7.6 for how to use the generated functions.

Note that for vectorization in `p`, the additional index `1` is necessary since `p` is a vectorized row vector (so, of shape $1 \times n_p \times n_{vec}$).

7.3.2. 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.

7.3.3. Automatic generation from symbolic expressions

Using the symbolic toolbox, the right-hand side and all of its derivatives can be generated automatically using the symbolic toolbox. For our example (19), this can be done by the commands (put into the separate script `gen_sym_demo1`) shown in Listing 3. The resuls are stored by the function `dde_sym2funcs` in a file **`sym_neuron.m`**, which can then be used without symbolic toolbox. The function `dde_sym2funcs` has the calling syntax

```
function [fstr,derivs]=dde_sym2funcs(f,x,par,...)
```

where `f`$\in \mathbb{R}^n$ is a symbolic expression containing symbols from the symbol arrays `x`$\in \mathbb{R}^{n\times(m+1)}$ and `par`$\in \mathbb{R}^p$. The arrays have to be of a form such that commands such as `jacobian(f,x(:))` and `jacobian(f,par)` are valid (producing symbolic jacobians). The output contains the string `fstr` that can be saved in a file as a matlab function and `derivs`, a structure with fields containing the symbolic derivatives. The outputs are usually not needed, since the typical way of storing the result is in a file by providing the name-value pair `'file',fname`, where `fname`. The output is then stored in the current folder as the function file **`fname.m`**. Important optional name-value pairs of arguments:

- `'file'`: filename where problem definition (right-hand side and delays if needed) are stored;
- `'sd_delay'`: $m \times 1$ symbol array of expressions for the delays for DDEs with state-dependent delays;
- `'directional_derivatives'` (logical, default `true`) should derivatives be stored as directional derivatives (set to `true` for problems with state-dependent delay).

```
dtanh =@(x)(1-tanh(x).^2);
ddtanh=@(x)2*(tanh(x).^2-1).*tanh(x);
neuron_sys_dirderi{1}=@(x,p,dx,dp)[...
 -dp(1,1,:).*x(1,1,:)-p(1,1,:).*dx(1,1,:)+...
  dp(1,2,:).*tanh(x(1,4,:))+p(1,2,:).*dtanh(x(1,4,:)).*dx(1,4,:)+...
  dp(1,3,:).*tanh(x(2,3,:))+p(1,3,:).*dtanh(x(2,3,:)).*dx(2,3,:);...
 -dp(1,1,:).*x(2,1,:)-p(1,1,:).*dx(2,1,:)+...
  dp(1,2,:).*tanh(x(2,4,:))+p(1,2,:).*dtanh(x(2,4,:)).*dx(2,4,:)+...
  dp(1,4,:).*tanh(x(1,2,:))+p(1,4,:).*dtanh(x(1,2,:)).*dx(1,2,:)];
neuron_sys_dirderi{2}=@(x,p,dx,dp)[...
 -2*dp(1,1,:).*dx(1,1,:)+...
  2*dp(1,2,:).*dtanh(x(1,4,:)).*dx(1,4,:)+p(1,2,:).*ddtanh(x(1,4,:)).*dx(1,4,:).^2+...
  2*dp(1,3,:).*dtanh(x(2,3,:)).*dx(2,3,:)+p(1,3,:).*ddtanh(x(2,3,:)).*dx(2,3,:).^2; ...
 -2*dp(1,1,:).*dx(2,1,:)+...
  2*dp(1,2,:).*dtanh(x(2,4,:)).*dx(2,4,:)+p(1,2,:).*ddtanh(x(2,4,:)).*dx(2,4,:).^2+...
  2*dp(1,4,:).*dtanh(x(1,2,:)).*dx(1,2,:)+p(1,4,:)*ddtanh(x(1,2,:))*dx(1,2,:).^2];
```

Listing 4: First two directional derivatives of right-hand side of eq. (19) (vectorized, see `../demos/neuron/html/demo1_funcs.html` and note that provision of this pair of functions is less complex than the routine for `neuron_sys_deri` required in previous versions.)

- `'multifile'` (logical, default `false`) store each function in a separate file. This may be necessary if complex expressions show up, which cause the code generation to create subfunctions (which may hae the same names in several instances). This will be noticed if the generated file is invalid.
- `'folder'` (character array, default `pwd()`): where file(s) should be stored.
- `'sys_tau_seq'` or `'sd_delay_seq'` (cell array, default empty) for state-dependent delays, indicate which index sequence of the delays can be called simultaneously. If the argument is empty, DDE-BIFTOOL assumes that it must call `sys_tau(i,x(:,1:i,:),p)` for $i = 1,\ldots,n_\tau$ in a sequential loop. If `'sys_tau_seq'` is, e.g.,`{1:3,4:7}` , then the calling sequence for the 7 delays will be

```
tau(1:3,:)=sys_tau(1:3,x(:,1,:),  p);
tau(4:7,:)=sys_tau(4:7,x(:,1:4,:),p);
```

7.3.4. Directional derivatives, manually provided

As an alternative to the automatic generation of derivatives the user may provide directional derivatives manually. The kth directional derivative of the right-hand side $f(x_0,\ldots,x_m,\eta)$,

$$d^k f(x_0,\ldots,x_m,\eta)[\tilde{x}_0,\ldots,\tilde{x}_m,\tilde{\eta}]^k := \frac{d^k}{dh^k} f(x_0+h\tilde{x}_0,\ldots,x_m+h\tilde{x}_m,\eta+h\tilde{\eta})|_{h=0},$$

has to have the form

```
function J=sys_dirderi(xx,par,dxx,dpar)
```

Arguments:

- `xx` $\in \mathbb{R}^{n\times(m+1)}$ contains the state variable(s), `xx(:,j)` $= x_{j-1}, x_0,\ldots,x_m$, 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);
- `dxx`$\in \mathbb{R}^{n\times(m+1)}$ contains the deviations that is applied to `xx`: `dxx(:,j)`$= \tilde{x}_{j-1}$ for $j = 1\ldots, m+1$(empty, one integer or two integers) index (indices) of `xx` with respect to which the right-hand side is to be differentiated
- `dpar`$\in \mathbb{R}^{1\times p}$ contains the deviations to the parameters, `dpar` $= \tilde{\eta}$.

The user passes on the set of all directional deriviatives in a $1 \times$ `maxorder` cell array to `set_funcs` (see section 7.6). For standard bifurcation analysis derivatives up to order 2 are used. For normal form analysis derivatives up to order 5 are used. Any derivatives that are not provided by either `sys_deri` or `sys_dirderi` will be approximated by finite differences. For the example (19), the first two directional derivatives can be implemented as given in Listing 4. Directional derivatives are easier to provide in vectorized form.

7.3.5. (No longer recommended) Derivatives or Jacobians of right-hand side — `sys_deri`

> **Recommended** course of action is to initially test the problem without manually provided derivatives. If one wants to perform normal form analysis or one encounters poor convergence for bifurcations, one should then try to automatically generate the right-hand side and its derivatives. See Section 7.3.3 for instructions how to automatically generate derivatives.
>
> The function described in this section is the most "manual" and most error-prone way of providing derivatives, still supported for backward compatibility. A simpler alternative is described as `sys_dirderi` in section 7.3.4, providing directional derivatives.

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 `dde_dirderiv`, combined with `dde_gen_deriv`. For speed-up or in case of convergence difficulties the user may provide the Jacobians of the right-hand side manually as a separate function. One way of specifying the Jacobian is to provide a function with a header 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.

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

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

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 $f \equiv f(x^0, x^1, \ldots, x^m, \eta)$. E.g., if nx has only one element take the derivative with respect to $x^{nx(1)}$. If it has two elements, take, of the result, the derivative with respect to $x^{nx(2)}$ and so on. Similarly, if np is nonempty take, of the resulting J, the derivative with respect to $\eta_{np(i)}$ where i ranges over all the elements of np, $1 \leqslant i \leqslant p$. 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,

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

with A_l as defined in (4).

The resulting routine is quite long, even for the small system (19); see file ../demos/neuron/html/neuron_sys_deri.html. Furthermore, implementing so many derivatives is an activity prone to a number of typing mistakes. For bifurcation analysis it is 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. For codimension-1 bifurcation tracking all other derivatives are only necessary in the Jacobians of the respective Newton procedures and thus influence only the convergence speed. Normal form analysis requires derivatives up to order 5.

Vectorization The function format also supports vectorization, such that the user should provide three-dimensional outputs J if the inputs xx is three-dimensional (and, if applicable, v is two-dimensional).

7.4. 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 7.3.2 (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 7.3 to reduce the computational effort.

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

$$\begin{aligned}
\frac{d}{dt}x_1(t) &= \frac{1}{p_1+x_2(t)}\left(1-p_2x_1(t)x_1(t-\tau_3)x_3(t-\tau_3)+p_3x_1(t-\tau_1)x_2(t-\tau_2)\right),\\
\frac{d}{dt}x_2(t) &= \frac{p_4x_1(t)}{p_1+x_2(t)}+p_5\tanh(x_2(t-\tau_5))-1,\\
\frac{d}{dt}x_3(t) &= p_6(x_2(t)-x_3(t))-p_7(x_1(t-\tau_6)-x_2(t-\tau_4))e^{-p_8\tau_5},\\
\frac{d}{dt}x_4(t) &= x_1(t-\tau_4)e^{-p_1\tau_5}-0.1,\\
\frac{d}{dt}x_5(t) &= 3(x_1(t-\tau_2)-x_5(t))-p_9,
\end{aligned} \tag{20}$$

where

$$\tau_1,\tau_2 \text{ are constant delays,} \quad \tau_3 = 2+p_5\tau_1x_2(t)x_2(t-\tau_1), \quad \tau_4 = 1-\frac{1}{1+x_1(t)x_2(t-\tau_2)},$$
$$\tau_5 = x_4(t), \qquad \tau_6 = x_5(t).$$

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`) for this system (20).

To define a system with state-dependent delays, the user should provide the MATLAB functions for right-hand sides and delays. The different ways of specifying the right-hand side are given in the following sections for system (20).

7.4.1. Right-hand side — `sys_rhs`

The definition and functionality of this routine is identical to the one described in section 7.3.1. 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 (20)) are also considered to be part of the parameters. See Listing 5 for the right-hand side to be provided for example (20). Vectorization will speed up computation for periodic orbits.

7.4.2. Delays — `sys_tau` and `sys_ntau`

The format and semantics of the routines specifying the delays differ from the one described in section 7.3.2. The user has to provide the functions,

```
function ntau=sys_ntau()
function  tau=sys_tau(ind,x,p)
function  tau=sys_dirdtau(ind,x,p,dx,dp) %optional, recommended
```

and the option

```
function f=sd_rhs(x,p)
%% Right-hand side of sd_demo (vectorized in x(1:5,1:7,:) and p(1,1:11,:))
f=NaN(size(x,1),size(x,3));
f(1,:)=(1./(p(1,1,:)+x(2,1,:))).*(1-p(1,2,:).*x(1,1,:).*x(1,4,:).*x(3,4,:)+...
    p(1,3,:).*x(1,2,:).*x(2,3,:));
f(2,:)=p(1,4,:).*x(1,1,:)./(p(1,1,:)+x(2,1,:))+p(1,5,:).*tanh(x(2,6,:))-1;
f(3,:)=p(1,6,:).*(x(2,1,:)-x(3,1,:))-p(1,7,:).*(x(1,7,:)-...
    x(2,5,:)).*exp(-p(1,8,:).*x(4,1,:));
f(4,:)=x(1,5,:).*exp(-p(1,1,:)*x(4,1,:))-0.1;
f(5,:)=3*(x(1,3,:)-x(5,1,:))-p(1,9,:);
end
```

Listing 5: Listing of right-hand side `'sys_rhs'` function for (20), here called `sd_rhs` (not vectorized).

```
...
'sys_tau_seq',{...}... % a cell array
```

to `set_funcs`, resulting in a cell array `funcs.sys_tau_seq`. The default for `'sys_tau_seq'` is `num2cell(1:ntau)`. The function `sys_ntau` has no arguments and returns the number of (constant and state-dependent) delays. For the example (20), this could be the anonymous function `@()6;`.

The function `sys_tau` has the three arguments:

- `ind=funcs.sys_tau_seq{k}` on kth call (integer(s) $\geqslant 1$) indicates, which delay(s) are to be returned; this will be determined by `funcs.sys_tau_seq{k}` on the kth call to `sys_tau`, and can be an array if `funcs.sys_tau_seq` indicates that `sys_tau` returns more than delay simultaneously;
- `x` ($n_x \times$ `ind(1)`-matrix) is the state with delays as determined so far: `x(:,1,:)`$= x(t)$, `x(:,k,:)`$= x(t-\tau_{k-1})$ for $k = 2 \ldots$ `ind(1)`;
- `par` (row vector) is the vector of system parameters

The output are the `funcs.sys_tau_seq{k}` delays $\tau_{\texttt{ind}}$. DDE-BIFTOOL calls `sys_tau` as often as there are elements in `funcs.sys_tau_seq` (so, maximal n_τ times if `funcs.sys_tau_seq` is `{1,2,...,ntau}`), where `ntau=sys_ntau()`. In the first call `xx` is the $n_x \times 1$ vector, $x(t)$ such that τ_1 may depend on $x(t)$ and `par`. After the first call to `sys_tau`, DDE-BIFTOOL computes $x(t-\tau_1), \ldots, x(t-\tau_j)$, if `funcs.sys_tau_seq{1}=1:j`. In the second call to `sys_tau`, `xx` is a $n_x \times (j+1)$ matrix, consisting of $[x(t), \ldots, x(t-\tau_j)]$ such that the delay τ_{j+1} may depend on $x(t)$,..., $x(t-\tau_j)$ and `p`, etc. In this way, the user can define state-dependent point delays with arbitrary levels of nesting, but calling `sys_tau` only as often as necessary if there are many delays (e..g, when approximating distributed delays). The delay function for the example (20) is given in Listing 6.

Implicitly defined delays Note that, due to the possiblility of specifying a singular matrix M in front of the left-hand side derivative $x'(t)$, it is in principle possible to specify fully implicit delays. One would introduce n_τ additional scalar variables x_{n+j} and add the n_τ equations

$$0 = x_{n+j}(t) - \tau_{\text{fun},j}(x(t), x(t-\tau_1), \ldots, x(t-\tau_{n_\tau}))$$

as part of the right-hand side `f` (`sys_rhs`). Then one may define the jth delay as equal to x_{n+j}: `sys_tau=@(i,x,p)x(n+i,1,:)` An example is the demo `poscontrol`, where the delay (travelling time) $s(t)$ is given by the implicit equation $cs(t) = x(t-s(t)) + x(t)$ (where c is the travelling speed and $x(t)$ is the distance from the reflector).

```
%% User-provided state-dependent delays for tutorial sd_demo:
function tau=sd_tau(k,x,p)
if all(k<3)
    tau=cat(1,p(1,10,:),p(1,11,:));
    tau=tau(k,:);
else
    tau=cat(1,...
        2+p(1,5,:).*p(1,10,:).*x(2,1,:).*x(2,2,:),...
        1-1./(1+x(2,3,:).*x(1,1,:)),...
        x(4,1,:),...
        x(5,1,:));
    tau=tau(k-2,:);
end
end
```

Listing 6: Listing of `'sys_tau'` function for (20), here called `sd_tau`.

Order of delays The order of the delays requested in `sys_tau` corresponds to the order in which they appear in `x` as passed to the functions `sys_rhs` and `sys_dirderi`.

Dependency of state-dependent delays on x The option `'sys_tau_seq'` of `set_funcs(...)` permits the user to specify which previously defined states $x(t),\ldots,x(t-\tau_{n_{\tau,j}})$ the delay τ_j depends on (default is $n_{\tau,j}=j-1$ for $j=1\ldots,n_\tau$). The option specifies a list of integer vectors, indicating, which delays can be computed simultaneously:

```
funcs=set_funcs(...'sys_tau_seq',{1:5},...);
```

indicates that $n_\tau=5$, and that DDE-Biftool may call `funcs.sys_tau` as

```
tau=funcs.sys_tau(1:5,x,p);
```

where x has format $n_x\times 1$. The option

```
funcs=set_funcs(...'sys_tau_seq',{1,2,3,4,5},...);
```

indicates that $n_\tau=5$, and that DDE-Biftool will call the delays one after another, since (e.g.) τ_4 may depend on $x(t-\tau_3)$:

```
for i=1:ntau
  tau(i)=funcs.sys_tau(i,x(:,1:i+1),p);
  % get x(t-tau(i)), insert into array x(1:nx,1:ntau)...
 end
```

Here, at step i, x has shape $n_x\times(i+1)$. If the system has many delays (e.g, when approximating distribted delays), the provision of `'sys_tau_seq'` increases execution speed. The built-in wrappers for distributed delay, `dde_add_dist_delay`, and `dde_sym_int_delay` call `set_funcs` with the appropriate option `'sys_tau_seq'`.

Difference to section 7.3.2 When calling `sys_tau` for a state-dependent delay, the *value* of the delay is returned. This is in contrast with the definition of `sys_tau` in section 7.3.2, where the position in the parameter list is returned.

```
ntau=6; % Set number of delays and parameter names
parnames=[strcat('p',num2cell('1':'9')),{'tau1','tau2'}];
x=sym('x',[5,ntau+1]);% Symbols for x, delays and parameters
syms(parnames{:});
par=sym(parnames);
%% Right-hand side
f=[(1/(p1+x(2,1)))*(1-p2*x(1,1)*x(1,4)*x(3,4)+p3*x(1,2)*x(2,3));...
   p4*x(1,1)/(p1+x(2,1))+p5*tanh(x(2,6))-1;...
   p6*(x(2,1)-x(3,1))-p7*(x(1,7)-x(2,5))*exp(-p8*x(4,1));...
   x(1,5)*exp(-p1*x(4,1))-0.1;...
   3*(x(1,3)-x(5,1))-p9];
%% Delay
delays=[tau1;                    tau2;   2+p5*tau1*x(2,1)*x(2,2);...
        1-1/(1+x(2,3)*x(1,1));  x(4,1);  x(5,1)];
%% Differentiate and generate code, exporting it to sym_sd_demo
dde_sym2funcs(f,x,par,'sd_delay',delays,'filename','sym_sd_demo');
```

Listing 7: Definition for right-hand side, delays and all derivatives of (20) using the symbolic toolbox. See Listing 5 for manually provided right-hand sides. See Section 7.6 for how to use the generated functions.

7.4.3. Automatic generation from symbolic expressions

Using the symbolic toolbox, the right-hand side, delays and all of their derivatives can be generated automatically using the symbolic toolbox. For our example (20), this can be done by the commands (put into the separate script **`gen_sym_sd_demo.m`**) shown in Listing 7. The resuls are stored by the function `dde_sym2funcs` in a file **`sym_sd_demo.m`**, which can then be used without symbolic toolbox. Note that the expression for the delay is passed on as name-value pair: `dde_sym2funcs(...,'sd_delay',tau,...)`.

7.4.4. Directional derivatives of delays

The user may provide directional derivatives for `sys_tau`. The kth directional derivative of the delay function $\tau_j(x_0,\dots,x_{j-1},\eta)$,

$$d^k\tau_j(x_0,\dots,x_{j-1},\eta)[\tilde{x}_0,\dots,\tilde{x}_{j-1},\tilde{\eta}]^k := \frac{d^k}{dh^k}\tau_j(x_0+h\tilde{x}_0,\dots,x_{j-1}+h\tilde{x}_{j-1},\eta+h\tilde{\eta})|_{h=0},$$

has to have the form

```
function J=sys_dirdtau(ind,xx,par,dxx,dpar)
```

Arguments (similar to section 7.3.4 for directional derivatives of the right-hand side):

- `ind` (integer $\geqslant 1$) the number of the delay,
- `xx` ($n \times$ `ind` matrix) is the state: `xx(:,1)` $= x(t)$, `xx(:,i)` $= x(t-\tau_{i-1})$ for $i = 2\dots$ `ind`;
- `par` $\in \mathbb{R}^{1\times p}$ are the parameters (as for the right-hand side)
- `dxx` ($n \times$ `ind` matrix) are the deviations $\tilde{x}_i$ from `xx`
- `dpar` $\in \mathbb{R}^{1\times p}$ contains the deviations to the parameters, `dpar` $= \tilde{\eta}$.

The user passes on the set of all deriviatives in a $1 \times$ `maxorder` cell array to `set_funcs` (see section 7.6). For standard bifurcation analysis derivatives up to order 1 are used. For normal form analysis derivatives up to order 5 are used. Any derivatives that are not provided by `sys_dirdtau` will be approximated by finite differences. Directional derivatives are easier to provide in vectorized form.

7.4.5. Directional derivatives of right-hand side

The definition and functionality of derivatives to the right-hand side are identical to the one described in Section 7.3. We do not present here the routine for `sys_dirderi` since it is quite long. See the symbolically generated derivatives in **`sym_sd_demo.m`** in the demo example **`sd_demo`**. If the user does not provide a function for the Jacobians the finite-difference approximation 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 and directional derivatives of `sys_rhs`.

7.5. Extra conditions — `sys_cond`

System routines `sys_cond` with a header of either of the types

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

can be used to add extra conditions during corrections and continuation, see section 11.2 for an explanation of arguments and outputs. There are several prepared functions available, e.g.,

1. `dde_extreme_cond` `dde_nlin_extreme_cond` for fixing extrema of `psol` profiles or nonlinear functions thereof,
2. `dde_stst_lincond`, `dde_psol_lincond` for fixing symmetries of `stst.x` or `psol.profile` arrays,
3. `sys_cond_coll_fixperiod` fixing period as equal to one of the parameters,
4. `dde_sys_cond_create` for creating general simple constraints.

If the argument `'sys_cond'` of `set_funcs` is a function, then which form DDE-BIFTOOL assumes is determined by another optional argument for `set_funcs`, `sys_cond_reference` (default `false` for first form, `true` for second form).

If one plans to pass on several `sys_cond` type conditions, one should convert the functions to structs that also contain basic information about arguments. E.g.,

```
u0psolcond=dde_sys_cond_create('name','u0fixpsol','fun',@(x)x,'args',{'profile',{1,1}});

u0psolcond =   struct with fields:
         name: 'u0fixpsol'
    reference: 1
          fun: @(p,pref)gen_cond(options.fun,p,pref,options.reference,
                  options.args,options.deriv,options.isvec)
```

is a `sys_cond` that fixes `profile(1,1)==0` (replacing a phase condition). The condition

```
sbxcond=dde_psol_lincond('sbx_symmetry',xdim,'x','trafo',Rsym,'shift',[1,2],...
  'condprojint',linspace(0,0.5,6)')

sbxcond =   struct with fields:
         name: 'sbx_symmetry'
    reference: 0
          fun: @(p)loc_psol_lincond(p,varargin{2:end})
```

enforces $R_{sym}x(t) - x(t+1/2)$ ($n_x = 2$ =`xdim`) at t =`linspace(0,0.5,6)`. Several structs of this type can be collected in an array for passing on to `set_funcs` for simultaneous enforcement.

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

> **Note:** see below for how to collect right-hand sides automatically generated from symbolc expressions into structure.

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 handles to all user-provided functions, the left-hand side matrix M and supporting information. 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 (19) of a neuron, discussed in section 7.3 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 7.3.1), and **`neuron_sys_deri.m`** is the filename in which the function providing the system derivatives are defined (see section 7.3.5). 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 `dde_gen_deriv`, is used):

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

For the sd-DDE example (20), 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 2 in section 8.1 later):

- `'sys_rhs'`: handle of the user function providing the right-hand side, described in sections 7.3.1 and 7.4.1;
- `'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 7.3.2, or providing the values of the delays for sd-DDEs as described in section 7.4.2;
- `'sys_dirderi'`: (default empty) cell array containing handles to directional derviatives of `'sys_rhs'`, described in sections 7.3.4 (alternative to `'sys_deri'`);
- `'sys_deri'`: handle of the user function providing the Jacobians of right-hand side, described in sections 7.3.5 and 7.4.5;
- `'sys_ntau'` (relevant for sd-DDEs only): handle of the user function providing the number of delays in sd-DDEs as described in section 7.4.2;
- `'sys_dirdtau'`: (relevant for sd-DDEs only, default empty) cell array containing handles to directional derviatives of `'sys_tau'`, described in sections 7.4.4 (alternative to `'sys_dtau'`);

- 'sys_dtau' (relevant for sd-DDEs only): handle of the user function providing the Jacobians of delays (for sd-DDEs), described in section 7.3.5;
- 'lhs_matrix' (numerical, default identity): the $n \times n$ matrix M on the left-hand side of the differential equation;
- 'sys_cond' (default empty): handle for user-specified extra conditions (see section 7.5 and section 11.2)
- 'sys_cond_reference': (logical, default false): flag indicating if 'sys_cond' takes extra argument, see section 7.5;
- 'x_vectorized' (logical, default false): if the functions in 'sys_rhs', 'sys_dirderi' 'sys_deri' (if provided), 'sys_tau' (for sd-DDEs) 'sys_dirdtau' and 'sys_dtau' (for sd-DDEs if provided) can be called with 3d arrays in their xx (and dxx) argument. Vectorization will speed up computations for periodic orbits.
- 'p_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 $1 \times p \times$ nvec arrays in their par (and dpar) argument. Vectorization will speed up computations for periodic orbits.

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.

7.6.1. Collection of problem definitions from dde_sym2funcs

If one has used dde_sym2funcs to generate right-hand sides, stored in a file (e.g., with name **sym_fcn.m**, which is an argument of dde_sym2funcs), the functions (or problem definitions) structure should be generated using set_symfuncs:

```
funcs=set_symfuncs(@sym_fcn,...);
```

Other arguments are given as name-value pairs from the argument list of set_funcs and override the defaults set inside set_symfuncs. In particular, sys_tau should be still set for problems with constant delay (but not for problems with state-dependent delay). For example, the problem definition for the example (19) after executing the symbolic expressions in Listing 3 is:

```
parnames={'kappa','beta','a12','a21','tau1','tau2','taus'};
cind=[parnames;num2cell(1:length(parnames))];
ip=struct(cind{:});
neuron_tau=@()[ip.tau1,ip.tau2,ip.taus];
funcs=set_symfuncs(@sym_neuron,'sys_tau',neuron_tau);
```

Note that the above listing creates a structure ip containing field storing the parameter indices, aiding with keeping track of parameter numbers

```
ip =   struct with fields:
    kappa: 1     beta: 2      a12: 3      a21: 4     tau1: 5     tau2: 6     taus: 7
```

for later use during bifurcation analysis. This type of structure avoids hard-coding parameter numbering.

7.7. Addition of approximate distributed delays

The `funcs` structure created by a single call to `set_funcs` is permitted to be "incomplete", that is, it may contain a function that depends on n_x dimensional x but returns only n_f dimensional output with $n_f < n_x$. This gradual problem build-up is useful for later addition of other equations. In particular, distributed delays are defined in the form of additional equations.

7.7.1. Simple integrals — example `renewal_demo`

Let us consider the renewal equation [8]

$$x(t) = \frac{\gamma}{2}\int_{\tau_2}^{\tau_1+\tau_2} x(t-s)(1-x(t-s))ds. \tag{21}$$

This example is illustrated in the demo `renewal_demo`. We note that, since the integrand does not depend on s, but only $t-s$, this equation can be transformed into a DDE with discrete delays (see file `renewal_disc_demo.m` in the `renewal_demo` folder) to compare results. Renewal equation (21) can be implemented in DDE-BIFTOOL, after rewriting it as a system by introducing the distributed delay as a new variable y:

$$0 = x(t) - \frac{\gamma}{2}y(t-\tau_2) \tag{22}$$

$$0 = \int_0^{\tau_1} x(t-s)(1-x(t-s))ds - y(t). \tag{23}$$

We implement the first equation like a standard DDE which, however, depends on 2 state variables:

```
[ix,iy,ig,itau1,itau2]=deal(1,2,1,2,3);
f_funcs=set_funcs('sys_rhs', @(x,p)x(ix,1,:)-p(1,ig,:).*x(iy,3,:)/2,...
    'sys_tau',@()[itau1,itau2],'lhs_matrix',[0,0],...
    'x_vectorized',true,'p_vectorized',true);
```

The specification of the left-hand side matrix M as `'lhs_matrix'` is mandatory for non-square `funcs` as it enables `set_funcs` to determine n_f. Before we can "add" the equation defining the distributed delay y, we need to create a structure that keeps track of which state variable indices, delays and parameter indices are used so far. This `funtab` structure needs information about the dimension of state x and parameter p (which `funcs` does not know). The easiest way to pass this information on is by passing on the initial guess:

```
par0([ig, itau1,itau2])=...
      [2,     2,    1];
u0(ix,1)=1-2/(par0(ig)*par0(itau1));
u0(iy,1)=u0(ix)*2/par0(ig);
tab_de=dde_add_funcs([],'rhs',f_funcs,'x',u0,'par',par0);
```

The call to `dde_add_funcs` takes a `funtab` structure (empty here), the `funcs` structure to be added after `'rhs'` and an example state (after `'x'`) and parameter vector (after `par`). It will internally perform a test call determining n_f. The function can be called again to add further equations. This is wrapped into `dde_add_dist_delay` for equations defining distributed delays and only depending on previously defined states and parameters of the following general form (14):

$$0 = \int_0^{\tau} g(s, x_i(t-s), p_\ell)ds - x_k. \tag{24}$$

The user provides an index i_τ (`'bound'`), and the index sets i (`'x'`), k (`'value'`) and ℓ (`'ipar'`) to `dde_add_dist_delay`.

```
g=@(s,x,p)x.*(1-x);
tab_dist=dde_add_dist_delay(tab_de,g,...
    'value',iy,'bound',itau1,'x',ix,'ipar',[],'int',4,'degree',3); %ix=1,iy=2,itau=1
funcs=dde_combined_funcs(tab_dist);
```

In this case i_τ =itau1, x =ix, k =iy and ℓ =[]. All indices may be non-scalar index vectors in general. The option 'bound_is_par' (default true) controls if the index i_τ refers to a state variable or a parameter. The first two arguments are the previously created tab_de and the integrand g. The integrand can be

- a vectorized function of (s, x, p) or (x, p) (controlled by option 'gt'),
- a cell array of function handles that contain g and its directional derivatives (up to order 2 are supported currently), e.g. [{g},dg], where in this case

```
dg = {@(s,x,p,ds,dx,dp)dx-2*x.*dx, @(s,x,p,ds,dx,dp)-2*dx.^2}
```

- the name or function handle of a function generated by dde_sym2funcs (see daphnia_demo).

The argument 'int' and 'degree' control the discretization. If 'int' is an integer it specificies the number of equally sized subintervals into which $[0, \tau]$ will be split for a piecewise polynomial approximation with polynomials of degree given by 'degree'. The option 'int' may also specify the mesh (scaled to $[0, 1]$) explicitly, e.g. ...'int',[0.1,0.25,0.5,1] to have a finer mesh for small $s \in [0, 1]$.

7.7.2. Chains of integrals

Since the index sets k and i in (24) can overlap and one may access delayed states x_k, the form (24) of distributed delays permits nested integrals and non-zero lower boundaries for the integrals (as demonstrated by (21)). To enable multiple integrals without introducing additional state variables, one may use a *chain* of nested integrals. A chain is added in the same way as dde_add_dist_delay but it needs separate *creation*, *adding* of integrals, and *closing* steps. The construction is motivated to enable treatment of problems such as the daphnia size structured population model [14], in the form presented in [1]. This section does not describe the equations in detail (see daphnia_demo with mathematical description and symbolically generated right-hand sides). The system is a coupled renewal and differential equation with state $x(t) = (r(t), \rho_A(t), b_d(t), c_d(t), s_{d,A}(t))$, where

- $r(t)$ is the resource (rescaled),
- $\rho_A(t)$ is the maturation age as a fraction of the maximum age a_{max},
- $b_d(t)$ the (rescaled) birth rate,
- $c_d(t)$ the consumption rate of resources by the entire poplulation,
- $s_{d,A}(t)$, the resource term in dinvidual growth.

The quantities $b_d(t)$, $c_d(t)$ and $s_{d,A}(t)$ will be distributed delays i nthe sense that they are determined as differences of intergrals over the past.

The resource r satisfies a differential equation of the form

$$\dot{r}(t) = f_r(r(t), c_d(t)). \tag{25}$$

The relative maturation age $\rho_A(t)$ is determined implicitly by the time it takes to grow to maturation size (a parameter) in an algebraic relation,

$$0 = f_{thr}(\rho_A(t), s_{d,A}(t)). \tag{26}$$

We create for the first two equations a 2×5 funcs structure, initialising a funtab structure tab_de in the same way as for simple integrals:

```
f_funcs=set_symfuncs(@sym_daphnia_dde,...
    'lhs_matrix',lhsmat); %lhsmat=[1,0,0,0,0; 0,0,0,0,0]
tab_de=dde_add_funcs([],'rhs',f_funcs,'x',u0,'par',par0); % u is 5x1, par is 1x12
```

Two integrals over the variable a (age) of the following form will be needed:

$$s_d(a,t) = \int_0^a g_{sd}(\alpha, r(t-\alpha), p_{sd})d\alpha \qquad \text{(effect of resource on growth)},$$
$$d_{eff}(a,t) = \int_0^a g_{deff}(\alpha, s_d(\alpha,t), b_d(t-\alpha), p_{deff})d\alpha \qquad \text{(cumulative population effect)}.$$

From these integrals one extracts the 3 "distributed delays" at the discrete ages a_{max} and $a_{max}\rho_A(t)$ by linear combinations:

$$b_d(t) = d_{eff}(a_{max},t) - d_{eff}(\rho_A(t)) \qquad \text{(scaled birth rate)} \tag{27}$$
$$c_d(t) = d_{eff}(a_{max},t) \qquad \text{(consumption)} \tag{28}$$
$$s_{d,A}(t) = s_d(a_{max}\rho_A(t),t) \qquad \text{(influence of resource on growth to maturity)}. \tag{29}$$

Equations (25)–(29) form the combined RE-DDE system. DDE-BIFTOOL puts the above into a general format by creating a sequence of variables $y_{id,0}(a,t)$ to $y_{int,2}(a,t)$ satisfying

$$\text{for } a\in[0,a_{max}] \quad y_{id,0}(a,t) = x_i(t-s) \quad \text{for } i=1,3 \text{ (so } x=(r,b_d)),$$
$$\text{for } a\in[0,a_{max}] \quad y_{id,1}(a,t) = g_{sd}(a,y_1(a,t),q_1) \quad \text{for } y_1(a,t) = y_{id,0,1}(a,t) = r(t-a),$$
$$\text{for } a\in[0,a_{max}] \quad y_{int,1}(a,t) = \int_0^a y_{id,1}(\alpha,t)d\alpha,$$
$$\text{for } a\in[0,a_{max}] \quad y_{id,2}(a,t) = g_{deff}(a,y_2(a,t),q_2) \quad \text{for } y_2(a,t) = (y_{int,1}(a,t), y_{id,0,2} = (s_d(a,t), b_d(t-a)),$$
$$\text{for } a\in[0,a_{max}] \quad y_{int,2}(a,t) = \int_0^a y_{id,2}(\alpha,t)d\alpha.$$

The parameters q_1 and q_2 in g_{sd} and g_{deff} are subsets of the full parameter set, namely $q_1 = p_{3,4}$, $q_2 = p_{1,4,5,7,8}$. As one can see, the chain of histories $y_{id,j}(a,t)$ and $y_{int,j}(a,t)$ starts with some components $x_i(t-a)$, then $y_{id,j}$ is a nonlinear function $g_j(a,y_j(a,t),p_j)$ of y_j, which are components of previously defined $y_{id,j}(a,t)$ and $y_{intj}(a,t)$, and $y_{int,j}(a,t)$ is the integral of $y_{id,j}(a,t)$ over a, starting from an initial value $y_{0,j}(t)$ (which can be 0, a state or a parameter). The corresponding code is

```
[ic_r,ic_bd]=deal(1,2);
taugrid=[0,0.05,0.1,0.15,0.2:0.1:1]; % mesh over history
idd=dde_create_chain_delay(ip.amax,[ix.r;ix.bd],...
    'grid',taugrid,'degree',4); % defines bound amax, y_{id,0} and grid
```

The first command `dde_create_chain_delay` *creates* the chain of integrals, specifying the bound (parameter number for `amax`), which x components enter $y_{id,0}$, and the mesh size/properties. The mesh is rescaled to $[0,1]$. the output `idd` is a structure containing bookkeeping information about the chain so far.

```
idd=dde_add_chain_delay(idd,'sd',@sym_sd_int,...
    'igx',ic_r,'igp',ip_sd); %ip_sd=[3,4,5]
```

The command `dde_add_chain_delay` defines $y_{id,1}$ and $y_{int,1}$. It passes on the previously opened chain `idd`, a name (`'sd'` for the chain element, the nonlinearity g_{sd} (`@sym_sd`), the index of $y_{id,0}$ (`'igx'`) and the parameters (`'igp'`) that g_{sd} depends on. If the argument `'igx'` is an integer it is assumed to refer to the index in $y_{id,0}$.

```
idd=dde_add_chain_delay(idd,'ceff',@sym_ceff_int,...
    'igx',{...
           'x' ,[ic_r,ic_bd],'id';... % y_id,0,1:2
           'sd',  1,         'int' ...% y_int,1,1
           },...
    'igp',ip_ceff);  %ip_ceff=[1,4,5,7,8]
```

The second command `dde_add_chain_delay` adds another chain element with name `'ceff'` and function `@sym_ceff_int`. This function depends on a seclection of previous y, which is now specified in their long form, in a 2×3 cell array, where the first column of each row is the name of the chain that is picked, the second is the index and the third column selects if the `'id'` or the `'int'` component is selected. In our case `@sym_ceff` depends a 3-dimensional y, namely $y_{id,0,1}$, $y_{id,0,2}$ and $y_{int,1,1}$, namely, $r(t-a)$, $b_d(t-a)$ and $s_d(a,t)$.

Finally, a closing step creates algebraic equations defining n_d distributed delays by combining selected components of the $y_{id,k}(a,t)$ and $y_{int,k}(a,t)$ at user-selected $r_k\tau_{max}$:

$$x_k = \sum_{k=1}^{n_d} S_{s,k} y_s(r_{s,k} a_{max}, t). \tag{30}$$

Here $y_s(r_k\tau_{max}, t)$ is a subvector of the sequence of $y_{id,k}(a,t)$ and $y_{int,k}(a,t)$. The code looks for the example like this.

```
tab=dde_close_chain_delay(tab,idd,...
  'ix',{...
         'sd',  1,'int';...
         'ceff',1,'int'
       },...
    'value',[ix.bd;ix.cd;ix.sdA],...
    'S',{[0,-1; 0,0; 1,0], [0,1; 0,1; 0,0]}, ...
    'it',[ix.raA,ip.amaxrel],'t_type',{'x','parameter'});
funcs=dde_combined_funcs(tab);
```

- The first argument `tab` the function `tab`.
- The second argument `idd` is the structure that stored the information about previously added chain elements.
- The argument `'ix'` selects the components of $y_{id,k}$ and $y_{int,k}$ used in (30). In this example, we select $y_{int,1}$ and $y_{int,2}$, which correspond to s_d and d_{eff}. The argument `'value'` lists the variable indices to which the distributed delays are assigned.
- The argument `'S'` is a cell array of the matrices $S_{s,k}$, defining the linear combinations resulting in the distributed delays.
- The argument `'it'` are the fractions $r_{s,k}$ at which the selected chain elements $y_{s,k}$ are evaluated.

The final command generates a `funcs` structure useable for bifurcation structure from `tab`.

8. 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.

field	content	purpose, default
`'sys_rhs'` **[!]**	function handle	right-hand side, function in file **`sys_rhs.m`** if found in current working folder
`'sys_ntau'`	function handle	number of delays for sd-DDEs, `@()0`
`'sys_tau'` **[!]**	function handle	delay indices or values, function in file **`sys_tau.m`** if found in current working folder
`sys_tau-seq`	cell array	calling order for delays, `num2cell(1:sys_ntau())`
`'sys_cond'`	function handle	extra conditions, `@(p)dummy_cond`, a provided routine that adds no conditions
`'sys_cond_reference'`	logical	`false`
(`'lhs_matrixfun'`)	function handle	`@(sz)eye(sz)`, set by passing on matrix to `'lhs_matrix'`
(`'sys_deri'`)	function handle	`dde_gen_deriv`, coming with DDE-BIFTOOL and using finite-difference approximation
`'sys_dirderi'`	cell array of function handles	directional derivatives of `sys_rhs`,`{}`
(`'sys_dtau'`)	function handle	derivative of `sys_tau`,`{}`
`'sys_dirdtau'`	cell array of function handles	directional derivatives of `sys_tau`,`{}`
`'x_vectorized'`	logical	are `sys_rhs`, `sys_tau` etc callable for `x` with shape `(nx,ntau+1,nvec)`?, `false`
`'p_vectorized'`	logical	are `sys_rhs`, `sys_tau` etc callable for `p` with shape `(1,np,nvec)`?,`false`
(`'tp_del'`)	logical	n/a (automatically determined from the number of arguments expected by `funcs.sys_tau`)
(`'wrap_...'`)	function handles	wrapped versions of `sys_...` (created automatically, can be passed on if vectorized)
`'drhs_dir'`	function handle	directional derivative of `wrap_rhs` up to arbitrary order
`'drhs_mf'`	function handle	multi-directional derivative of `wrap_rhs` up to arbitrary order, permits expansions such as `{1,'I'},0` to generate jacobians
(`'is_lhs_matrix_set'`)	logical	n/a (set automatically)
(`'sys_dtau_provided'`)	logical	n/a (set automatically)
(`'dirderi_provided'`)	logical	n/a (set automatically)
(`'sys_dirtau_provided'`)	logical	n/a (set automatically)

Table 2: **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 7.6. Only fields marked with **[!]** are mandatory arguments of `set_funcs` (for sd-DDEs `'sys_ntau'` is mandatory, too).

8.1. Problem definition (functions) structure

The user-provided functions described in Section 7 get passed on to DDE-BIFTOOL's routines collected in a single argument `funcs`, a structure containing at least the fields listed in Table 2.

Only fields marked with **[!]** are mandatory (for sd-DDEs `'sys_ntau'` is mandatory, too). The user does 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 7.6. See also the tutorial demos `neuron` and `sd_demo` (see `../demos/index.html` for examples of usage.

> **Warning:** The structure may contain more fields for some problems. In particular, some constructors generate problem structures for extended systems from the user-defined problem structure. These may contain auxiliary fields for managing the link between extended system and original system. Hence, the user is expected to create a complete problem structure by calls to provided routines such as `set_funcs` or constructors (such as `SetupPOfold`).

8.2. Point structures

Table 3 describes the structures used to represent a single steady state, fold, Hopf, periodic and homoclinic/heteroclinic solution point. If, as a user one wants to create a variable `pt` of a certain point type the recommended function to use is

```
pt=dde_kind_create(...)
```

where `kind` is replaced by the particular type of point (e.g., `stst`, `fold`, `hopf`, `psol`, `hcli`). The arguments of `dde_kind_create` are name value pairs, similar to the `struct(...)` function. However, `dde_kind_create` function performs additional sanity checks, inserts defaults (for example, providing the field `'kind'` is not necessary) and puts the fields in the order described in Table 3. For a point structure `pt` one may call the functions

```
ind=dde_ind_from_point(pt,free_par_ind)
x=  dde_x_from_point(pt,free_par_ind)
pt= dde_point_from_x(x,ptemplate,free_par_ind)
```

to convert between the point structure and a numerical vector of variables used during continuation and equation solving. The output `ind` of `dde_ind_from_point` returns a structure of indices, pointing to the location of the variable entries in `x`.

Steady states and their bifurcation points A steady state solution is represented by the parameter values p (which contain also the delay values for the constant-delay case, see section 7) and x^*. A fold bifurcation is represented by the parameter values p, its position x^* and a null-vector of the characteristic matrix $\Delta(0)$. A Hopf bifurcation is represented by the parameter values p, its position x^*, a frequency ω and a (complex) null-vector of the characteristic matrix $\Delta(i\omega)$.

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

$$t_{(i-1)d+j} = t_{(i-1)d+1} + b_j(t_{id+1} - t_{(i-1)d+1}).$$

The *scaled internal representation points* $b_j \in [0,1]$ are values that can be set by the user, but are assumed to be independent of the subinterval i.

> *Warning*: this assumption is periodically checked using the function `dde_coll_check`. It needs to be fulfilled for correct results!

field	content
'kind'	'stst'
'parameter'	$\mathbb{R}^{1\times p}$
'x'	$\mathbb{R}^{n\times 1}$
'stability'	[] or struct
'nmfm'	[] or struct
'nvec'	[] or struct
'flag'	char array

(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'	[] or struct
'nmfm'	[] or struct
'nvec'	[] or struct
'flag'	char array

(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'	[] or struct
'nmfm'	[] or struct
'nvec'	[] or struct
'flag'	char array

(c) Steady state Hopf

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

(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 3: **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^+.

The profile is a continuous piecewise polynomial on the mesh. More specifically, it is a polynomial of degree d on each subinterval $[t_{(i-1)d+1}, t_{id+1}]$ for $i = 1, \ldots, L$. The polynomial q_i on interval $[t_{(i-1)d+1}, t_{id+1}]$ is uniquely represented by its values at the points $\{t_{(i-1)d+j} : j = 1, \ldots, d+1\}$. Hence the complete profile is represented by its values at all the mesh points,

$$x^*(t_\ell), \ \ell = 1, \ldots, Ld+1.$$

Because polynomials on adjacent intervals share the value at the common interval point t_{id+1} for $i = 0, \ldots L$, this representation is automatically continuous (it is, however, not ensured to be continuously differentiable).

The default choice for the scaled internal representation points b_j are equidistant,

$$b_j = \frac{j-1}{d} \quad j = 1, \ldots, d,$$

for reasons of backward compatibility to version < 3.2 (this has also been the choice in AUTO, MatCont and the COLL toolbox of COCO as of 2019). This representation provides stable interpolation only for low degrees d (error amplification is of order 2^d [6]). If one plans to use high-order interpolation (e.g., $d > 10$), the user should choose representation nodes of orthogonal polynomials. A pre-defined choice are Chebyshev nodes of the second kind. The default representation nodes are Chebyshev nodes if a degree $d > 10$ is chosen (through `p_topsol`, `SetupPsol` or `dde_psol_from_...`), if the left-hand side matrix M is singular, or if the point method parameter `'collocation_parameters'` equals `'force_smooth'` (ensuring differentiability at mesh points).

The function

```
y=dde_coll_eva(point,t,...)
```

returns values of the profile at arbitrary points `t` inside `point.mesh([1,end])` in `y`. If the optional argument `'output','matrix'` is provided it returns the matrix J such that `J*point.profile(:)` equals `reshape(x(t)[],1)`. Other options include `'diff'` to obtain derivatives, and `'submesh_limit'` to select which one-sided limits are taken when `'diff'` option is non-zero.

Connecting orbits 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, However, the current implementation does only permit equidistant representaion points $b - j$.

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.

8.3. 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 computed stability information in the form described in Table 4. This information depends on the type of computation method chosen.

With default method settings 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. Alternatively one may request the eigenvalues closest to a particular complex value (see section 8.4 below).

field	content
'h'	[] or $\mathbb{R}$
'l0'	$\mathbb{C}^{n_l}$
'l1'	$\mathbb{C}^{n_c}$
'n1'	[] or $\{-1, 0, \ldots\}^{n_c}$
'err'	$\mathbb{C}^{1\times n_l}$
'v'	[] or $\mathbb{C}^{n\times n_l}$
'w'	[] or $\mathbb{C}^{n\times n_l}$
'discarded'	[] or struct

(a) Structure in field 'stability' for steady state, fold and Hopf points of Table 3

field	content
'mu'	$\mathbb{C}^{n_m\times 1}$
'eigenfuncs'	[] or struct^{n_m}
'err'	$\mathbb{R}^{1\times n_m}$

(b) Structure in field 'stability' for periodic orbit points of Table 3

Table 4: **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.

Discretizations 'mxo' and 'bdf' 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 corrected roots presents more accurate yet less robust information than the collection of approximate roots, see section 11. The field 'v' contains right eigenvectors for corrected roots. The field 'err' contains the error in the characteristic equation $\max|\Delta(\lambda)v|$ for corrected roots. If correction was performed and some roots were discarded then the discarded roots are stored in a struct in the field 'discarded.l0', along with the error 'discarded.err'.

Discretization 'cheb' (default) The field 'l0' stores the eigenvalues, the fields 'v', 'w' store right and left eigenvectors ($\Delta(\lambda)v = 0$, $w^H\Delta(\lambda) = 0$, scaling $w^H\Delta'(\lambda)v = 1$). The field 'err' contains the error estimate $\max|\Delta(\lambda)v|$ for each eigenvalue. The field 'l1' has entries identical to 'l0' since no correction is done. The fields 'n1' and 'h' are empty.

For periodic solutions only approximations to the Floquet multipliers are provided in a field 'mu' (in order of decreasing modulus). If the method field 'geteigenfuncs' is set then the field 'eigenfuncs' contains an array of points of kind 'psol' with the eigenfunctions corresponding to 'mu'. As the characteristic matrix is not analytically available, DDE-BIFTOOL does not offer an additional correction. If 'geteigenfuncs' was set to true, then the field 'err' is computed. This field contains the residuals of the discretized matrix eigenvalue problem, so is underestimating the true error.

> **Warning:** A common source of error in Floquet multipliers is that the corresponding eigenfunction is rapidly oscillating but the mesh is sufficiently fine only for the (non-oscillating) periodic orbit. So, it is advisable to set 'geteigenfunc' to true and inspect the eigenfunctions to understand unexpected results.

8.4. Method parameters

Part of DDE-BIFTOOL are three types of numerical algorithms: continuation, nonlinear equation solving and approximation of eigenvalues of the linearized right-hand side in solutions. The `'method'` field of a solution branch contains structures defining method parameters for these algorithms in its fields `'continuation'`, `'point'` and `'stability'`. A `'method'` field with default settings is output by the function `df_mthod`. The function is called with a character array indicating the point type (`'kind'`) of points in the branch as a single input (e.g. `df_mthod('stst')`).

field	content	default value
`'newton_max_iterations'`	$\mathbb{N}_0$	`5, 5, 5, 5, 10`
`'newton_nmon_iterations'`	$\mathbb{N}$	`1`
`'preprocess'`	fcn name	`''`
`'postprocess'`	fcn name	`''`
`'extra_condition'`	$\{0,1\}$	`0`
`'print_residual_info'`	$\{0,1\}$	`0`
`'jacobian_nonsquare'`	$\{0,1\}$	`0`
`'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_columns'`	logical	`false`
*`'phase_condition'`	$\{0,1\}$	`1`
*`'collocation_parameters'`	$[0,1]^d$ or empty	empty
*`'remesh'`	$\{0,1\}$	`1`
*`'delay_zero_prep'`	point type	`'stst'` or `'coll'`
*`'adapt_mesh_before_correct'`	$\mathbb{N}$	`0`
*`'adapt_mesh_after_correct'`	$\mathbb{N}$	`3`
*`'matrix'`	`'full'` or `'sparse'`	`'full'`

Table 5: **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.

Nonlinear equation solving — the `'point'` field 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 5.

For the computation of periodic solutions, additional fields are necessary, marked with an asterisk (*) in Table 5. The meaning of the different fields in Table 5 is explained in section 11.

Parameters controlling the pseudo-arclength continuation (using secant approximations for tangents) are stored in a structure of the form given in Table 7. 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 6.

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 6: **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
`'warnings'`	logical	`true`
`'warn_angle'`	$[-1,1)$	`-1`
`'use_tangent'`	logical	`false`
`'permit_negative_delay'`	logical	`false`
`'halt_before_reject'`	$\{0,1\}$	`0`
`'stops'`	array of structs	`empty`

Table 7: **Continuation method structure**: fields and possible values.

8.5. 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	subfield	content	
'point'		array of points	(s. Table 3)
'method'	'point'	point method struct	(s. Table 5)
	'stability'	stability method struct	(s. Table 6)
	'continuation'	continuation method struct	(s. Table 7)
'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 8: **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 8.

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 5. The second, called 'stability', contains stability method parameters as described in Table 6 and the third, called 'continuation', contains continuation method parameters as described in Table 7. 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 7.3 (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.

8.6. 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 9. 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 9. 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

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

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').

9. 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,varargin)
```

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 3).
- 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 3). 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 5).
- 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, use this point as reference for extra conditions and (e.g.) the phase condition. Thus, the phase shift is minimized with respect to this point, if `method.point.phase_condition` is true. 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.
- `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).

Name-value pairs in `varargin` are passed on to `method` fields.

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

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 3).
- `method`: a stability method structure (see table 6).
- `stability`: the computed stability of the point through a collection of approximated eigenvalues (as a structure described in table 4). 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.

Name-value pairs in `varargin` are passed on to `method` fields.

```
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 '$*$'.

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 8) 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 9).

```
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): `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 $\tau_{tz} = 0$ and $d\tau_{tz}/dt = 0$.

10. 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,varargin)
```

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 8.
- `max_tries`: maximum number of steps allowed.
- useful name-value pair: `'ax'` or `'plotaxis'` and an axis handle blistax. This ensures that the online plotting, if requested, occurs in in axis `ax`, instead of the most recently visited axis `gca` (which is default).
- All other name-value pairs are passed on to `branch.method` and its fields.
- `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 9).

```
function [st_branch,nunst,dom]=br_stabl(funcs,branch,varargin)
```

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 8).
- `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.
- additional arguments:
  - If next argument is numeric: `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.
  - If next argument is numeric: `recompute`; if zero, do not recompute stability information present. If nonzero, discard and recompute old stability information present (for points which were not skipped).

- Other arguments are name-value pairs. The pairs `'skip'` and `'recompute'` have the same meaning as above and are an alternative way to pass on the arguments.
- `'exclude_trivial'` (logical, `true`) exclude eigenvalues and Floquet multipliers thatare known to be on the imaginary axis or unit circle from the count for `nunst`.
- `'locate_trivial'` function handle `@(p)...`. Function that returns the trivial eigenvalues for point `p`. The default depends on the kind of point. For `'fold'` one eigenvalue 0 is excluded, for `'hopf'`, the imaginary pair closest to `1i*p.omega*[-1,1]` is excluded, for `'psol'` the Floquet multiplier closest to 1 is excluded (measured via `log`). For `'psol'` where `funcs` argument indicates that they are periodic orbit bifurcations, the correponding additional critical Floquet multipliers are excluded. If solutions have additional continuous symmetry, additional trivial eigenvalues may have to be excluded.
- `pointtype_list` (function handle, default `@pointtype_list`), overrides defaults for eigenvalue exclusions.
- `'exclude'` (function handle, `@(p,z)false(size(z))`, provide criteria for eigenvalues that should be ignored for stability count (e.g., high-frequency imaginary eigenvalues for renewal equations).
- Other arguments are passed on to `branch,method.stability`.
- `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`).
- `nunst`: number of unstable eigenvalues, excluding trivial ones and those meeting the criterion passed on in `'exclude'`.

```
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 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 8).
- `x_measure`: (scalar) measure to produce plotting quantities for the x-axis (see table 9). If empty, the point number is used to plot against.
- `y_measure`: (scalar) measure to produce plotting quantities for the y-axis (see table 9). 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 8) 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 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 8).
- `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 8).
- `measure`: given measure (see table 9).
- `col`: the collection of measures taken along the branch (over its point array) ordered rowwise. 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.

11. 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 [41, 22, 21, 20, 23, 46] or in [19]. For details on applying these methods to bifurcation analysis of sd-DDEs see [40].

11.1. 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.

$$f(x^*, x^*, \ldots, x^*, \eta) = 0. \tag{31}$$

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.

$$\begin{aligned} 0 &= f(x^*, x^*, \ldots, x^*, \eta) \\ 0 &= \Delta(x^*, \eta, 0)v \\ 0 &= c^T v - 1 \end{aligned} \tag{32}$$

(see (6) for the definition of the characteristic matrix Δ). Here, $c^T v - 1 = 0$ presents a suitable normalization of v. The vector $c \in \mathbb{R}^n$ is chosen as $c = v^{(0)}/({v^{(0)}}^T v^{(0)})$, 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.

$$\begin{aligned} 0 &= f(x^*, x^*, \ldots, x^*, \eta) \\ 0 &= \Delta(x^*, \eta, i\omega)v \\ 0 &= c^H v - 1 \end{aligned} \tag{33}$$

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.

$$\begin{aligned} \dot{u}(c_{i,j}) &= Tf\left(u(c_{i,j}), u\left(\left[c_{i,j} - \frac{\tau_1}{T}\Big|_{\mathrm{mod}[0,1]}\right], \ldots, u\left(\left[c_{i,j} - \frac{\tau_m}{T}\Big|_{\mathrm{mod}[0,1]}\right], \eta\right)\right), \\ &\qquad i = 0, \ldots, L-1,\ j = 1, \ldots, d \\ 0 &= u(0) - u(1) \bmod (2\pi), \\ p(u) &= 0. \end{aligned} \tag{34}$$

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

$$\int_0^1 \dot{u}_{\mathrm{ref}}(s)u(s)ds = 0, \tag{35}$$

where u is the current solution and u_{ref} is a reference point (e.g., initial guess or previous point along the branch). The collocation points are obtained as

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

from the interval points t_i, $i = 0, \ldots, L-1$ and the collocation parameters c_j, $j = 1, \ldots, d$. The profile u is discretized as a piecewise polynomial as explained in section 8. This representation has a discontinuous derivative at the interval points. If $c_{i,j}$ coincides with t_i the right derivative is taken in (34), 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.

$$\begin{aligned}
\dot{u}(c_{i,j}) =& Tf(u(c_{i,j}), u(c_{i,j} - \frac{\tau_1}{T}), \ldots, u(c_{i,j} - \frac{\tau_m}{T}), \eta) = 0, \quad (i = 0, \ldots, L-1,\ j = 1, \ldots, d) \\
u(\tilde{c}) =& x^- + \epsilon \sum_{k=1}^{s^-} \alpha_k v_k^- e^{\lambda_k^- T \tilde{c}}, \qquad \tilde{c} < 0 \\
0 =& f(x^-, x^-, \eta) \\
0 =& f(x^+, x^+, \eta) \\
0 =& \Delta(x^-, \lambda_k^-, \eta) v_k^- \\
0 =& c_k^H v_k^- - 1, \quad (k = 1, \ldots, s^-) \\
0 =& \Delta^H(x^+, \lambda_k^+, \eta) w_k^+ \\
0 =& d_k^H w_k^+ - 1, \quad (k = 1, \ldots, s^+) \\
0 =& {w_k^2}^H (u(1) - x^+) + \sum_{i=1}^{G} g_i {w_k^+}^H e^{-\lambda_k^+(\theta_i + \tau)} A_1(x^+, \eta) \left(u(1 + \frac{\theta_i}{T}) - x^+ \right), \quad (k = 1, \ldots, s^+) \\
u(0) =& x^- + \epsilon \sum_{i=1}^{s^-} \alpha_k v_k^- \\
1 =& \sum_{i=1}^{s^-} |\alpha_k|^2 \\
0 =& p(u, \eta)
\end{aligned} \tag{36}$$

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 (33).

Point method parameters The point method parameters (see table 5) 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 (§11.2).
- `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 5) provide the following information.

- `phase_condition`: when nonzero the integral phase condition (35) 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.

11.2. 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 5). 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 (19) subject to the following extra conditions

$$\begin{aligned} T &= 200, \\ 0 &= a_{12}^2 + a_{21}^2 - 1, \end{aligned} \tag{37}$$

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 8 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 8: Implementation extra conditions (37) using a routine `sys_cond`.

If one wants to add several functions of type `sys_cond`, one should use the function

```
function [cond,name]=dde_sys_cond_create(...)
```

The name-value paris are

- `'name'` (`char`), name used to dientify the condition
- `'fun'` (function handle of type `@(p)...` or type `@p,pref`,
- `'reference'` (logical, default `false`) indicates if function needs also reference point argument `pref`,

One may use this function also to create simple extra conditions directly. For example, the above `sys_cond` may be created directly with the command

```
cond=dde_sys_cond_create('name','fixparT','fun',@(a,b,T)[T-1;a.^2+b.^2-1],...
  'args',{'parameter',{1,3},'parameter',{1,4},'period',1});
```

11.3. 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 (31), (32), (33), (34) or (36) 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.

A new field `'use_tangent'` enables using the tangent prediction instead of secant prediction and permits limiting the angle (or, rather `cos(angle)` between predictor tangent and secant between corrected point and previous point.

Continuation method parameters The continuation method parameters (see table 7) 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.
- `use_tangent` (logical, default `false`) use tangent instead of secant prediction. This option enables `'minimal_angle'` enforcement.
- `'minimal_angle'` (real in $[-1, 1]$, default -1) cos of angle between tangent predictor and secant between corrector and reference point. Only enforced if `'use_tangent'` is `true`.
- `plot_measure`: if empty use default measures to plot. Otherwise `plot_measure` contains two fields, 'x' and 'y', which contain measures (see table 9) 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.

11.4. Roots of the characteristic equation

Roots of the characteristic equation are approximated using a linear multi-step (LMS-) method applied to (3).

Consider the linear k-step formula

$$\sum_{j=0}^{k} \alpha_j y_{L+j} = h \sum_{j=0}^{k} \beta_j f_{L+j}. \tag{38}$$

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

$$\tilde{y}(t_j + \epsilon h) = \sum_{l=-r}^{s} P_l(\epsilon) y_{j+l},\ \epsilon \in [0,1). \tag{39}$$

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 (38) 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 (3) around a steady state solution $x^*(t) \equiv x^*$ we have

$$f_j = A_0 y_j + \sum_{i=0}^{m} A_i \tilde{y}(t_j - \tau_i) \tag{40}$$

where we have omitted the dependency of A_i on x^*. The stability of the difference scheme (38), (40) 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 (38) 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 (7),

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

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

$$\mathrm{Re}(\lambda) = \frac{\ln(|\mu|)}{h}. \tag{41}$$

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

$$\mathrm{Im}(\lambda) \equiv \frac{\arcsin(\frac{\mathrm{Im}(\mu)}{|\mu|})}{h} \pmod{\frac{\pi}{h}}. \tag{42}$$

For small h, $0 < h \ll 1$, the smallest representation in (42) is assumed the most accurate one (that is, we let arcsin map into $[-\pi/2, \pi/2]$).

The parameters r and s (from formula (39)) are chosen such that $r \leqslant s \leqslant r + 2$ (see [34]). The choice of h is based on the related heuristic outlined in [23].

Approximations for the rightmost roots λ obtained from the LMS-method using (41), (42) can be corrected using a Newton process on the system,

$$\begin{aligned} 0 &= \Delta(\lambda)v \\ 0 &= c^T v - 1 \end{aligned} \tag{43}$$

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 (43) 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 4 (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 6 (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 [19, §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, $\frac{h}{\tau} \geqslant$ `minimal_time_step`.
- `maximal_time_step`: maximal time step relative to maximal delay, $\frac{h}{\tau} \leqslant$ `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 $\mathrm{Re}(\lambda) \geqslant$ `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 (43).
- `root_accuracy`: required accuracy of the norm of the residual of (43) 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.

11.5. 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 (34) 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 4 (right). Approximations to the Floquet multipliers are kept in field `mu`.

Stability method parameters The stability method parameters (see table 6 (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.

12. 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 (2) and (9). 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 do not claim fitness for purpose) 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.

12.1. 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 $\exp(At)f(x_0,\ldots,x_m) = f(\exp(At)x_0,\ldots,\exp(At)x_m)$).
- 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 [11] (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* [4] 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] Alessia Ando et al. *Collocation methods for complex delay models of structured populations*. PhD thesis, Università degli Studi di Udine, 2020.

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

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

[4] 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.

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

[6] Jean-Paul Berrut and Lloyd N Trefethen. Barycentric lagrange interpolation. *SIAM review*, 46(3):501–517, 2004.

[7] 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.

[8] Dimitri Breda, Odo Diekmann, Davide Liessi, and Francesca Scarabel. Numerical bifurcation analysis of a class of nonlinear renewal equations. *Electronic Journal of Qualitative Theory of Differential Equations*, 2016(65):1–24, 2016.

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

[10] 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.

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

[12] 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.

[13] 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.

[14] Odo Diekmann, Mats Gyllenberg, Johan AJ Metz, Shinji Nakaoka, and Andre M de Roos. Daphnia revisited: local stability and bifurcation theory for physiologically structured population models explained by way of an example. *Journal of mathematical biology*, 61(2):277–318, 2010.

[15] 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.

[16] 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.*

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

[18] 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.

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

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

[21] 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.

[22] 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.

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

[24] 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.

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

[26] Leon Glass and Michael Mackey. Mackey-glass equation. *Scholarpedia*, 5(3):6908, 2010.

[27] Leon Glass and Michael C Mackey. Pathological conditions resulting from instabilities in physiological control systems. *Annals of the New York Academy of Sciences*, 316(1):214–235, 1979.

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

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

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

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

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

[33] 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.

[34] 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.

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

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

[37] 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.

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

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] C. Rackauckas and Q. Nie. DifferentialEquations.jl - a performant and feature-rich ecosystem for solving differential equations in Julia. *Journal of Open Research Software*, 5(1):15, 2017. http://doi.org/10.5334/jors.151.

[45] 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.

[46] 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.

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

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

[49] 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.

[50] 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.

[51] J. Sieber and B. Krauskopf. Bifurcation analysis of an inverted pendulum with delayed feedback control near a triple-zero eigenvalue. *Nonlinearity*, 17(1):85–104, 2004.

[52] Jan Sieber. Local bifurcations in differential equations with state-dependent delay. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 27(11):114326, 2017.

[53] 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.

[54] 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.

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

A. GNU Octave compatibility considerations

`nargin` incompatibility The core DDE-BIFTOOL code of version v2.0x was likely `GNU Octave` compatible. The changes to 3.1, replacing function names by function handles, broke this compatibility initially, because, for example, the call `nargin(sys_tau)` gives an error message in `GNU 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 8.1 and Table 2).

As of version 3.8.1, `GNU 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 `GNU 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 `GNU 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 `GNU Octave` does not appear to experience this slow-down.

Useful options to be set in `GNU 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/`