The Brain Dynamics Toolbox for Matlab

Stewart Heitmann, Matthew J Aburn, Michael Breakspear
QIMR Berghofer Medical Research Institute
300 Herston Road, Herston QLD 4006, Australia

Abstract

Nonlinear dynamical systems are increasingly informing both theoretical and empirical branches of neuroscience. The Brain Dynamics Toolbox provides an interactive simulation platform for exploring such systems in MATLAB. It supports the major classes of differential equations that arise in computational neuroscience: Ordinary Differential Equations, Delay Differential Equations and Stochastic Differential Equations. The design of the graphical interface fosters intuitive exploration of the dynamics while still supporting scripted parameter explorations and large-scale simulations. Although the toolbox is intended for dynamical models in computational neuroscience, it can be applied to dynamical systems from any domain.

Keywords: initial-value problems, differential equations, numerical integration, visualization, brain dynamics

---

1. Introduction

Computational neuroscience relies heavily on numerical methods for simulating non-linear models of brain dynamics. Software toolkits are the manifestation of those endeavors. Each one represents an attempt to balance mathematical flexibility with computational convenience. Toolkits such as GENESIS [1], NEURON [2] and BRIAN[3] provide convenient methods to simulate conductance-based models of single neurons and networks thereof. The Virtual Brain [4] scales up that approach to the macroscopic dynamics of the whole brain by combining neural field models [5] with anatomical connectivity datasets [6]. Mathematical toolkits such as AUTO [7], XPPAUT [8], MATCONT [9], PyDSTool [10] and CoCo [11] are useful for analyzing non-linear dynamics but assume advanced mathematical theory.

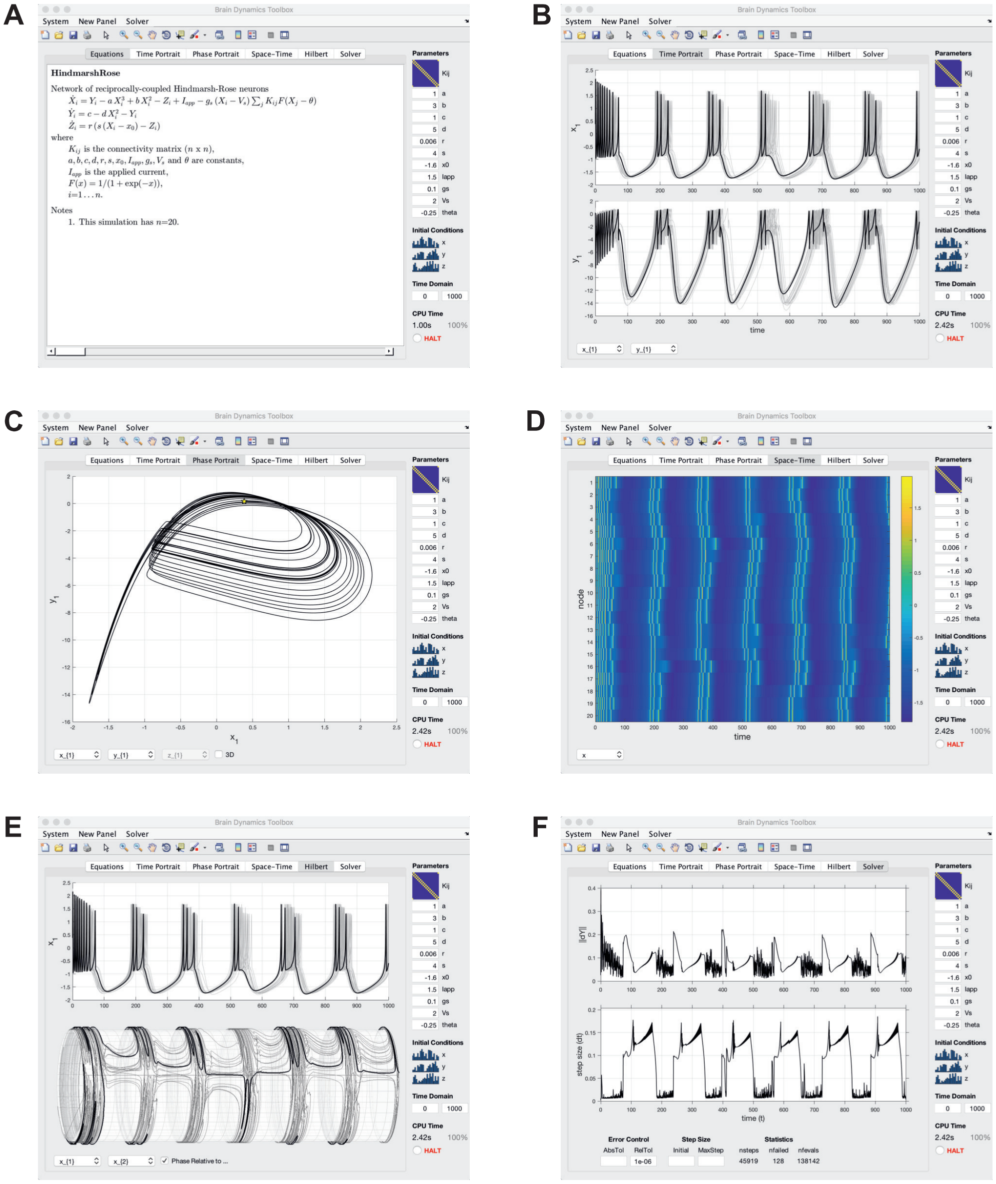
2. Problems and Background

In our experience, the existing computational toolkits often present technical barriers to broader audiences in cognitive neuroscience, systems neuroscience and neuroimaging. For example, GENESIS [1], NEURON [2], BRIAN [3] and XPPAUT [8] each use idiosyncratic languages for defining the differential equations. The Virtual Brain [4], CoCo [11] and PyDSTool [10] use conventional programming languages (Python and MATLAB) but assume advanced object-oriented programming techniques that broader audiences often find confusing. Of all of the existing toolkits, only XPPAUT [8] and the Virtual Brain [4] are capable of supporting Ordinary Differential Equations (ODEs), Delay Differential Equations (DDEs) and Stochastic Differential Equations (SDEs). Our Brain Dynamics Toolbox aims to bridge these technical barriers by allowing those with diverse backgrounds to explore neuronal dynamics through phase space analysis, time series exploration and other methods with minimal programming burden. A custom system of ODEs, DDEs or SDEs can typically be implemented in fewer than 100 lines of standard MATLAB code. Object-oriented programming techniques are not required. Once the model is implemented, it can be run interactively in the graphical interface (Figure 1) where a variety of different plotting panels and numerical solvers can be applied with no additional programming effort. The internal states of the graphical interface are accessible to the user's workspace so that parameter sweeps can be semi-automated in the command window with simple for-loop statements. Additional command-line tools are also provided for scripting fully-automated simulations in batch mode. Such scripts may be called from third-party MATLAB applications and vice versa. Large-scale simulations can be scripted to run in parallel using the MATLAB Parallel Computing Toolbox or the MATLAB Distributed Computing Server. Unfortunately the toolbox does not run on *Octave* [12] because of incompatibilities in the graphical interface class libraries.

3. Software Framework

The toolbox operates on user-defined systems of ODEs, DDEs and SDEs. The details differ slightly for each type of differential equation but the overall approach is the same. For an ODE,

$$\frac{dY}{dt} = F(t, Y),$$



HindmarshRose

Network of reciprocally-coupled Hindmarsh-Rose neurons

$$\dot{X}_i = Y_i - a\,X_i^3 + b\,X_i^2 - Z_i + I_{app} - g_s\,(X_i - V_s)\sum_j K_{ij}F(X_j - \theta)$$
$$\dot{Y}_i = c - d\,X_i^2 - Y_i$$
$$\dot{Z}_i = r\,(s\,(X_i - x_0) - Z_i)$$

where

K_{ij} is the connectivity matrix (n x n),
$a, b, c, d, r, s, x_0, I_{app}, g_s, V_s$ and θ are constants,
I_{app} is the applied current,
$F(x) = 1/(1+\exp(-x))$,
i=1 … n.

Notes

1. This simulation has n=20.

Parameters
1 a
3 b
1 c
5 d
0.006 r
4 s
-1.6 x0
1.5 Iapp
0.1 gs
2 Vs
-0.25 theta

Time Domain
0 1000

CPU Time
1.00s 100%

CPU Time
2.42s 100%

Error Control
AbsTol
RelTol 1e-06

Step Size
Initial
MaxStep

Statistics

nsteps	nfailed	nfevals
45919	128	138142

Figure 1: Screenshots of selected display panels in the graphical interface as it simulates a network of n=20 Hindmarsh-Rose [13] neurons. The parameters of the model appear in the control panel on the right-hand side of the application window. The solution is automatically recomputed each time any of those controls are altered. Individual controls can be scalar, vector or matrix values thereby accommodating arbitrarily large parameter sets. **A** Mathematical equations rendered with LaTeX. **B** Time portraits. **C** Phase portrait. **D** Space-time portrait. **E** Hilbert transform. **F** Solver step sizes.

the right-hand side of the equation is implemented as a matlab function of the form `dYdt=F(t,Y)`. The toolbox takes a handle to that function and passes it to the relevant solver routine on the user's behalf. The solver repeatedly calls `F(t,Y)` in the process of computing the evolution of $Y(t)$ from a given set of initial conditions. The toolbox uses the same approach as the standard MATLAB solvers (e.g. `ode45`) except that it also manages the input parameters and plots the solver output. To do so, it requires the names and values of the system parameters and state variables. Those details (and more) are passed to the toolbox via a special data structure that we call a *system structure*. It encapsulates everything needed to simulate a user-defined model. Once a system structure has been constructed, it can be shared with other toolbox users.

3.1. Software Architecture and Functionality

The hub-and-spoke software architecture (Figure 2) allows arbitrary combinations of solver routines and display panels to be applied to any model. The modular design also allows new solver routines and display panel classes to be added to the toolbox incrementally. The list of numerical solver routines and graphical panels that the toolbox supports continues to grow rapidly. The current version (2017c) supports the standard ODE solvers (`ode45, ode23, ode113, ode15s, ode23s`) and DDE solver (`dde23`) that are shipped with MATLAB. As well as a fixed-step ODE solver (`odeEul`) and two SDE solvers (`sdeEM, sdeSH`) that are specific to the Brain Dynamics Toolbox. The two SDE solvers are specialized for stochastic equations that use Itô calculus and Stratonovich calculus respectively.

The display panels can be used to visualize the dynamics, compute metrics from the time-series, or transform them into new time-series. The toolbox currently includes display panels for rendering mathematical equations, time plots, phase portraits, space-time plots, computing linear correlations, Hilbert transforms, surrogate data transforms and inspecting the individual steps taken by the solvers. The panel outputs are themselves accessible to the user's workspace as read-only variables. New panels can be added to the toolbox at any time and we encourage advanced users to write custom panels for their own projects although that level of graphical interface development does involve object-oriented programming.

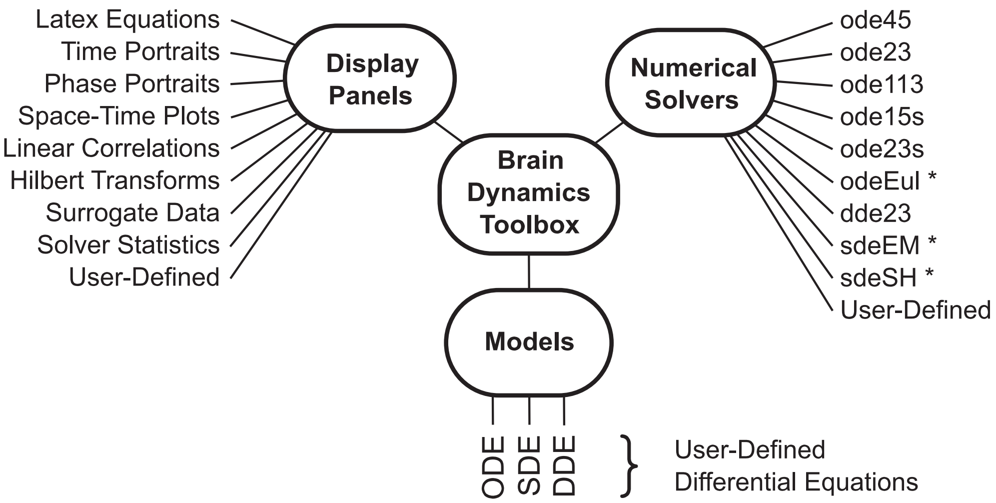


Figure 2: The hub-and-spoke software architecture of the Brain Dynamics Toolbox. Numerical solvers marked with an asterisk are unique to the toolbox.

4. Illustrative Example

We demonstrate the implementation of a network of recurrently-connected Hindmarsh-Rose [13] neurons,

$$\dot{X}_i = Y_i - a\,X_i^3 + b\,X_i^2 - Z_i + I_i - I_i^{net}, \tag{1}$$

$$\dot{Y}_i = c - d\,X_i^2 - Y_i, \tag{2}$$

$$\dot{Z}_i = r\,(s\,(X_i - x_0) - Z_i), \tag{3}$$

where X_i is the membrane potential of the i^{th} neuron, Y_i is the conductance of that neuron's excitatory ion channels, and Z_i is the conductance of its inhibitory ion channels. Each neuron in the network is driven by a locally applied current I_i and a network current $I_i^{net} = g_s\,(X_i - V_s)\sum_j K_{ij}F(X_j - \theta)$ that represents the synaptic bombardment from other neurons. The sigmoidal function $F(x)=1/(1+\exp(-x))$ transforms that synaptic bombardment to an equivalent ionic current. The connectivity matrix K_{ij} defines the weightings of the synaptic connections between neurons. All other parameters in the model are scalar constants. The model is a typical example of a neuronal network as a system of coupled ODEs.

4.1. Defining the equations

We define the right-hand side of equations (1–3) as Matlab function of the form `dY=F(t,Y,...)` where the vector `Y` contains the instantaneous values of $[X(t), Y(t), Z(t)]$ at time t. The ellipses denote model-specific parameters.

```
% The ODE function for the Hindmarsh Rose model.
function dY = odefun(t,Y,Kij,a,b,c,d,r,s,x0,I,gs,Vs,theta)
    % extract incoming variables from Y
    Y = reshape(Y,[],3);        % reshape Y to (nx3)
    x = Y(:,1);                 % x is (nx1) vector
    y = Y(:,2);                 % y is (nx1) vector
    z = Y(:,3);                 % z is (nx1) vector

    % The network coupling term
    Inet = gs*(x-Vs) .* Kij./(1+exp(-x+theta));

    % Hindmarsh-Rose equations
    dx = y - a*x.^3 + b*x.^2 - z + I - Inet;
    dy = c - d*x.^2 - y;
    dz = r*(s*(x-x0)-z);

    % return result (3n x 1)
    dY = [dx; dy; dz];
end
```

It is no coincidence that the form of this function is identical to that required by the standard ODE solvers, since the toolbox applies those same solvers to this function. In order to do so, it requires the names and initial values of the model's parameters and state variables to also be defined. That is the purpose of the model's system structure.

4.2. Defining the system structure

The system structure (named `sys` by convention) encapsulates the function handles and parameter settings that the toolbox needs to pass the user-defined ODE function to the solver and plot the solution that is returned. The most important fields of the structure are the handle to user-defined function (`sys.odefun`), the names and initial values of the system variables (`sys.vardef`) and the names and values of the system parameters (`sys.pardef`). Once a system structure has been constructed, it can be saved to a *mat* file and used by the toolbox as is. Nonetheless it is common practice to provide a helper function that constructs a new system structure for a particular configuration of the model. In this example, the configuration of the system variables depends on the size of the network connectivity matrix, `Kij`.

```matlab
% Construct a system structure for the Hindmarsh-Rose model
function sys = HindmarshRose(Kij)
    % Infer the number of neurons from the size of Kij
    n = size(Kij,1);

    % Handle to our ODE function
    sys.odefun = @odefun;

    % Our ODE parameters
    sys.pardef = [ struct('name','Kij',   'value',Kij);
                   struct('name','a',     'value',1);
                   struct('name','b',     'value',3);
                   struct('name','c',     'value',1);
                   struct('name','d',     'value',5);
                   struct('name','r',     'value',0.006);
                   struct('name','s',     'value',4);
                   struct('name','x0',    'value',-1.6);
                   struct('name','Iapp',  'value',1.5);
                   struct('name','gs',    'value',0.1);
                   struct('name','Vs',    'value',2);
                   struct('name','theta', 'value',-0.25) ];

    % Our ODE variables
    sys.vardef = [ struct('name','x', 'value',rand(n,1));
                   struct('name','y', 'value',rand(n,1));
                   struct('name','z', 'value',rand(n,1)) ];

    % Latex (Equations) panel
    sys.panels.bdLatexPanel.title = 'Equations';
    sys.panels.bdLatexPanel.latex = {
        '\textbf{HindmarshRose}';
        '';
        'Network of coupled Hindmarsh-Rose neurons';
        '\qquad $\dot X_i = Y_i - a\,X_i^3 + b\,X_i^2 - Z_i +
    I_{app} - g_s\,(X_i-V_s) \sum_j K_{ij} F(X_j-\theta)$';
        '\qquad $\dot Y_i = c - d\,X_i^2 - Y_i$';
        '\qquad $\dot Z_i = r\,(s\,(X_i-x_0) - Z_i)$';
        'where';
        '\qquad $K_{ij}$ is the connectivity matrix,';
        '\qquad $a, b, c, d, r, s, x_0, I_{app}, g_s, V_s$
   and $\theta$ are constants,';
```

```
40         '\qquad $I_{app}$ is the applied current,';
41         '\qquad $F(x) = 1/(1+\exp(-x))$,';
42         '\qquad $i{=}1 \dots n$.' };
43 end
```

The order of the parameter definitions in the `pardef` field must match that of the `odefun` function. Likewise for the system variables in the `vardef` field.

The final part of the helper function (lines 28–42) defines the model-specific strings for rendering the mathematical equations in the LaTeX display panel. Those LaTeX strings are important for documenting the model in the graphical interface but they play no part in the simulation itself.

4.3. Running the model.

The model is run by loading an instance of the system structure into the toolbox graphical user interface, which is called `bdGUI`.

```
>> n = 20;                         % Define number of neurons.
>> Kij = circshift(eye(n),1) ...   % Define connection matrix,
       + circshift(eye(n),-1);     % as a chain in this case.
>> sys = HindmarshRose(Kij);       % Construct the sys struct.
>> bdGUI(sys);                     % Run the model in the GUI.
```

The graphical interface (Figure 1) allows the solution to be visualized with any number of display panels, all of which are updated concurrently. The solution is automatically recomputed whenever any of the graphical controls are adjusted; including the system parameters, the initial conditions of the state variables, the time domain of the simulation and the solver options.

4.4. Controlling the model

The `bdGUI` application returns a handle to itself which can be used to control the simulation from the MATLAB command window.

```
>> gui = bdGUI(sys)
gui =
  bdGUI with properties:
    version: '2017c'           % toolbox version string
        fig: [1x1 Figure]      % application figure handle
        par: [1x1 struct]      % system parameters (read-write)
       var0: [1x1 struct]      % initial conditions (read-write)
        var: [1x1 struct]      % solution variables (read-only)
```

```
          t: [1x9522 double]   % solution time points (read-only)
        lag: []                % DDE lag parameters (read-write)
        sys: [1x1 struct]      % system structure (read-only)
        sol: [1x1 struct]      % solver output (read-only)
        sox: []                % auxiliary variables (read-only)
     panels: [1x1 struct]      % display panel outputs (read-only)
```

The parameters of the model are all accessible by name via the `gui.par` structure. Likewise, the computed solution variables are accessible by name via the `gui.var` structure and also in the native format returned by the solver via the `gui.sol` structure. Parameter values written into the `gui.par` handle are immediately applied to the graphical user interface, and vice versa. Hence it is possible to use workspace commands to orchestrate parameter sweeps in the graphical user interface. For example, the workspace command

```
>> for r=linspace(0.05,0.001,25); gui.par.r=r; end;
```

sweeps the r parameter (time constant of inhibition) from 0.05 to 0.001 in 25 increments. The graphical interface automatically recomputes the solution every time that `gui.par.r` is assigned a new value in the loop. The result is an animated sequence of simulations where bursting phenomenon is observed for $r \lesssim 0.01$.

4.5. Scripting the model

The toolbox also provides a small suite of command-line tools for running models without invoking the graphical interface. Of these, the most notable commands are `bdSolve(sys,tspan)` which runs the solver on a given model for a given time span; and `bdEval(sol,t)` which interpolates the solution for a given set of time points.

```
>> t = 0:1000;
>> sol = bdSolve(sys,[t(1) t(end)]);
>> X = bdEval(sol,t);
>> plot(t,X);
```

The `bdEval` function is equivalent to the MATLAB `deval` function except that it also works for solution structures (`sol`) returned by third-party solvers.

5. Conclusions

The Brain Dynamics Toolbox provides researchers with an interactive graphical tool for exploring user-defined dynamical systems without the burden of programming bespoke graphical applications. The graphical interface imposes no limit the size of the model nor the number of parameters involved. System parameters and variables can range in size from simple scalar values to large-scale vectors or matrices without loss of generality. The design also imposes no barrier to scripting large-scale simulations and parameter surveys. The toolbox is aimed at students, engineers and researchers in computational neuroscience but it can also be applied to general problems in dynamical systems. It is supported with an extensive user manual [14] that provides detailed instructions for implementing new systems of ODEs, DDEs and SDEs. Once a new model is implemented, it can be readily shared with other toolbox users. The toolbox thus serves as a hub for sharing models as much as it serves as a tool for simulating them.

Acknowledgements

MATLAB® is a registered trademark of The Mathworks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098 USA, 508-647-7000, Fax 508-647-7001, *info@mathworks.com, www.mathworks.com*

[1] J. M. Bower, D. Beeman, The book of Genesis: exploring realistic neural models with the General Neural Simulation System., Telos, Springer, New York, 1998.

[2] N. T. Carnevale, M. L. Hines, The NEURON book, Cambridge University Press, 2006.

[3] D. F. M. Goodman, R. Brette, BRIAN simulator, Scholarpedia 8 (1) (2013) 10883. doi:10.4249/scholarpedia.10883.

[4] V. Jirsa, O. Sporns, M. Breakspear, G. Deco, A. R. McIntosh, Towards the virtual brain: network modeling of the intact and the damaged brain, Archives Italiennes de Biologie 148 (3) (2010) 189–205.

[5] V. K. Jirsa, H. Haken, Field theory of electromagnetic brain activity, Physical Review Letters 77 (5) (1996) 960.

[6] R. Kotter, Online retrieval, processing, and visualization of primate connectivity data from the CoCoMac database, Neuroinformatics 2 (2) (2004) 127–144.

[7] E. J. Doedel, A. R. Champneys, T. F. Fairgrieve, Y. A. Kuznetsov, B. Sandstede, X. Wang, AUTO 97: Continuation and bifurcation software for ordinary differential equations (with HomCont) (1998).

[8] B. Ermentrout, Simulating, analyzing, and animating dynamical systems: a guide to XPPAUT for researchers and students, SIAM, 2002.

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

[10] R. Clewley, Hybrid Models and Biological Model Reduction with PyDSTool, PLOS Computational Biology 8 (8) (2012) e1002628. doi:10.1371/journal.pcbi.1002628.

[11] H. Dankowicz, F. Schilder, Recipes for Continuation, SIAM, 2013.

[12] J. W. Eaton, GNU Octave Manual, Network Theory Limited, 2002.

[13] J. L. Hindmarsh, R. M. Rose, A model of neuronal bursting using three coupled first order differential equations, Proceedings of the Royal Society of London B: Biological Sciences 221 (1222) (1984) 87–102.

[14] S. Heitmann, M. Breakspear, Handbook for the Brain Dynamics Toolbox: Version 2017c, 1st Edition, QIMR Berghofer Medical Research Institute, 2017.

Required Metadata

Current executable software version

Nr.	**(executable) Software metadata description**	**Please fill in this column**
S1	Current software version	2017c
S2	Permanent link to executables of this version	https://github.com/breakspear/bdtoolkit/releases/tag/bdtoolkit-2017c
S3	Legal Software License	BSD 2-clause
S4	Computing platform/Operating System	Matlab 2014b or newer
S5	Installation requirements & dependencies	Signal Processing Toolbox (optional). Statistics and Machine Learning Toolbox (optional).
S6	If available, link to user manual - if formally published include a reference to the publication in the reference list	http://www.bdtoolbox.org
S7	Support email for questions	heitmann@bdtoolbox.org

Table 1: Software metadata (optional)

Current code version

Nr.	**Code metadata description**	**Please fill in this column**
C1	Current code version	2017c
C2	Permanent link to code/repository used of this code version	https://github.com/breakspear/bdtoolkit/releases/tag/bdtoolkit-2017c
C3	Legal Code License	BSD 2-clause
C4	Code versioning system used	git
C5	Software code languages, tools, and services used	Matlab 2014b or newer
C6	Compilation requirements, operating environments & dependencies	Signal Processing Toolbox (optional). Statistics and Machine Learning Toolbox (optional).
C7	If available Link to developer documentation/manual	http://www.bdtoolbox.org
C8	Support email for questions	heitmann@bdtoolbox.org

Table 2: Code metadata (mandatory)