BDEtools: A MATLAB package for Boolean delay equation modelling

Ozgur E. Akman[1*], Kevin Doherty[1,2] and Benjamin J. Wareham[1]

1. Department of Mathematics, The University of Exeter,

North Park Road, Exeter EX4 4QE, U.K.

2. Now with: CitySwift, CityPoint, 13-27 Prospect Hill, Galway H91 P9KP, Ireland

*Corresponding author

ABSTRACT

Boolean Delay Equations (BDEs) can simulate surprisingly complex behaviour, despite their relative simplicity. In addition to steady state dynamics, BDEs can also generate periodic and quasiperiodic oscillations, m:n frequency locking and even chaos. Furthermore, the enumerability of Boolean update functions and their compact parametrisation means that BDEs can be leveraged to generate low-level descriptions of biological networks, from which more detailed formulations (*e.g.* differential equation models) can be constructed. However, although several studies have demonstrated the utility of BDE modelling in computational biology, a current barrier to the wider adoption of the BDE approach is the absence of freely-available simulation software.

In this work, we present BDEtools – an open-source MATLAB package for numerically solving BDE models. After giving a brief introduction to BDE modelling, we describe the package's solver algorithms, together with several utility functions that can be used to provide solver inputs and to process solver outputs. We also demonstrate the functionality of BDEtools by illustrating its application to an established model of a gene regulatory network that controls circadian rhythms.

BDEtools makes it straightforward for researchers to quickly build reliable BDE models of biological networks. We hope that its ease of use and free availability will encourage more researchers to explore BDE formulations of their systems of interest. Through the continued use of BDEs by the computational biology community, we will no doubt discover their potential applicability to a broader class of biological networks.

1. INTRODUCTION

Many real world phenomena can be modelled by switch-like behaviour, with variables that are either on or off – even those that are typically modelled using continuous variables. For example, the activities of many genes are considered to essentially be either above a critical threshold for action or below this threshold. Indeed, although gene regulatory networks (GRNs) are usually modelled using ordinary differential equations (ODEs) or delay differential equations (DDEs), the continuous model outputs can exhibit switch-like behaviour (Wynn *et al.*, 2012; Montefusco *et al.*, 2016; Steinacher *et al.*, 2016).

Boolean delay equations (BDEs) offer an alternative modelling framework in which model variables are represented logically, being equal either to 1 (ON – suprathreshold) or 0 (OFF – subthreshold). Although discrete in state space, they are continuous in time. This can be contrasted with modelling

approaches that are continuous in state and time (*e.g.* ODEs, DDEs), continuous in state but discrete in time (*e.g.* difference equations) and discrete in both state and time (*e.g.* cellular automata) (Ghil *et al.*, 2008).

In BDEs, physical processes are described by the delay it takes for a change in one variable to impact another. In other words, BDE parameters describe the timescales associated with physical processes, rather than – for example – the kinetic constants used in ODE models. This results in simpler descriptions of biological systems with fewer parameters, facilitating the model calibration process. The representation of state variables as logical values means that complicated equations describing variable interactions, in an ODE for instance, are reduced to simpler expressions that involve logical operators (Kauffman *et al.*, 2004; Nikolajewa *et al.*, 2007; Akman *et al.*, 2012; Doherty *et al.*, 2017; Akman and Fieldsend, 2020; Berg *et al.*, 2021).

BDEs, therefore, hold significant promise as a simulation tool that can be incorporated into the mathematical modeller's toolbox. Where the mechanisms underlying biological systems cannot be determined from observation, yet the timescales of processes are apparent, BDEs are a natural choice. They can also be used as an initial step in the modelling process, for example to determine a particular network architecture, before a more detailed model is developed (Akman *et al.*, 2012; Wynn *et al.*, 2012; Tokuda *et al.*, 2019; Foo *et al.*, 2020). However, we are not currently aware of any existing open-source software designed specifically for solving or analysing BDEs, and as a result, we have developed the BDEtools package for the MATLAB programming environment.

2. BACKGROUND

2.1. BDEs: an overview

The main concepts underlying the BDE formalism were first developed in detail by René Thomas in the 1970s to provide a compact description of GRN behaviour (Thomas, 1973, 1978). Several studies have subsequently leveraged the BDE formalism to construct reduced GRN models (Thomas, 1991; Öktem *et al.*, 2003; Watterson *et al.*, 2008; Yu *et al.*, 2008; Sevim *et al.*, 2010; Watterson and Ghazal, 2010; Akman *et al.*, 2012; Doherty *et al.*, 2017; Akman and Fieldsend, 2020; Alyahya *et al.*, 2021). BDE models have also been used in other fields, including endocrinology (Berg *et al.*, 2021), climatology (Ghil *et al.*, 1987; Wright *et al.*, 1990; Saunders and Ghil, 2001), seismology (Zaliapin *et al.*, 2003a,b), economics (Coluzzi *et al.*, 2011) and electrical circuitry (Zhang *et al.*, 2009). The papers by Dee & Ghil (Dee and Ghil, 1984) and Ghil & Mulhaupt (Ghil and Mullhaupt, 1985) are good references for some fundamental results on the mathematical properties of BDE systems.

In BDEs, the states of system variables x_i are represented by logical values that are either ON ($x_i = 1$) or OFF ($x_i = 0$) at each time value, t. A system of BDEs with n variables is written as

$$x_i(t) = f_i(x_1(t - \tau_{i1}), x_2(t - \tau_{i2}), \ldots, x_n(t - \tau_{in})), \tag{1}$$

for $1 \leq i \leq n$, where $x_i(t) \in \{0, 1\}$ denotes the state of the ith variable at time t and τ_{ij} is the signalling delay that prescribes the time it takes for x_j to affect x_i. We refer to the functions $f_i : \{0, 1\}^n \to \{0, 1\}$ as *logic gates*. These functions specify how the interactions between $\{x_1, \ldots, x_n\}$ determine the state of x_i. Given sufficient knowledge of the system being modelled, the f_is may be fixed. However, an advantage of BDE modelling is that these functions can be easily parametrised (Akman *et al.*, 2012). Hence, in its most general form, the BDE system (1) is parametrised by both the set of delays $\boldsymbol{\tau} = (\tau_{11}, \ldots, \tau_{nn})$

and the collection of logic gates $\{f_1, \ldots, f_n\}$. To obtain a solution $\{\mathbf{x}(t) = (x_1(t), \ldots, x_n(t)) : t \geq t_0\}$ of (1) for a given combination of logic gates and delays, it also necessary to specify an *initial history*, $\mathbf{x}_h = \{\mathbf{x}(t) : t_0 - t_h \leq t \leq t_0\}$, where $t_h \geq \max(\boldsymbol{\tau})$.

Following Dee & Ghil (Dee and Ghil, 1984), we find it convenient to define the *memorisation variables*, $x_{ij}^M(t)$, where

$$x_{ij}^M(t) = x_i(t - \tau_{ij}), \quad i = 1, \ldots, n, \quad j = 1, \ldots, n, \tag{2}$$

that is, the values of the memorisation variables $\{x_{ij}^M(t)\}$ for x_i at time t are the values of the variable x_i at times $t - \tau_{ij}$, for $j = 1, ..., n$. It is therefore equivalent to say that the current state is calculated using the states of the variables at previous times, or using the current state of the memorisation variables, and it is the latter calculation that is utilised by the BDEtools solvers.

2.2. A BDE model with an analytical solution

Fig. 1(a) shows a circuit diagram for a two-component negative feedback loop, in which the gene B activates the gene A with a delay of one time unit and the gene A inhibits the gene B with a delay of one time unit. The BDEs that model this simple circuit are

$$A(t) = B(t - 1), \quad B(t) = \neg A(t - 1), \tag{3}$$

where $\neg$ denotes logical negation (NOT). By direct substitution, we obtain

$$A(t) = \neg A(t - 2),\ B(t) = \neg B(t - 2) \text{ and } A(t) = A(t - 4),\ B(t) = B(t - 4).$$

Assuming the history

$$\begin{aligned} A(t) &= 0, \quad \text{if } -1 < t \leq 0, \\ B(t) &= 1, \quad \text{if } -1 < t \leq 0, \end{aligned} \tag{4}$$

it follows that the solution to eqns. (3) for $t > 0$ is

$$\begin{aligned} A(t) &= \begin{cases} 1, & \text{if } 1 < \mathrm{mod}(t, 2) \leq 3, \\ 0, & \text{otherwise,} \end{cases} \\ B(t) &= \begin{cases} 0, & \text{if } 2 < \mathrm{mod}(t, 2) \leq 4, \\ 1, & \text{otherwise.} \end{cases} \end{aligned} \tag{5}$$

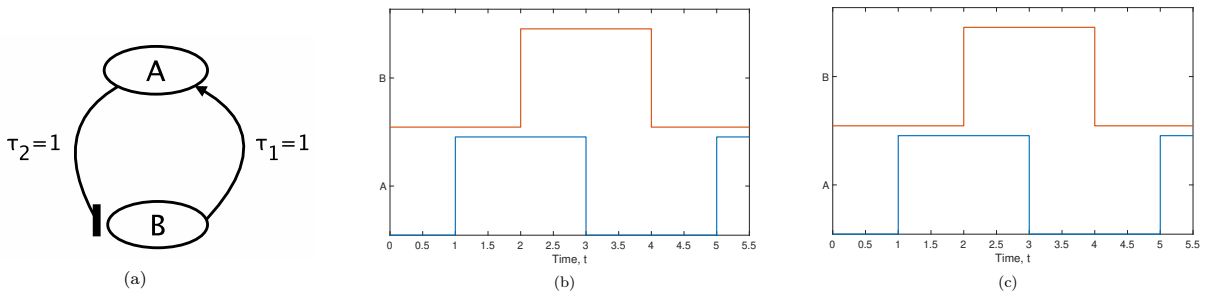
BDE systems with analytical solutions – such as the example given above – can be leveraged to assess the performance of numerical solvers, as we will see in section 3.3.

3. OVERVIEW OF THE BDETOOLS PACKAGE

3.1. The main BDE solver routine: `bdesolve`

The key BDEtools algorithm is a solver, bdesolve, that given a function describing the model equations, a set of delays, a history and a timespan will return the switch points and associated variable values over the specified range. Specifically, for an initial history

$$\mathbf{x}_h(t) = \left(x_1^h(t), \ldots, x_n^h(t)\right) : t_0 - t_h \leq t \leq t_0$$



(a) (b) (c)

FIG. 1: (a) Schematic of the simple negative feedback model (3), where an arrow represents activation and a blunted arrow represents inhibition. **(b)** Numerical solution `nsol` to (3) over $0 < t \leq 5.5$, computed from the initial history $\{A(t) = 0,\, B(t) = 1;\, -1 < t \leq 0\}$ using the routine `bdesolve`. The values of A and B after each switch in `nsol.x` are given by each column in `nsol.y`. **(c)** The corresponding analytical solution `asol`.

and a projection end time t_E, `bdesolve` generates a set of Boolean timeseries $\{\hat{\mathbf{x}}(t) = (\hat{x}_1(t), \ldots, \hat{x}_n(t)) : t_0 \leq t \leq t_E\}$ that satisfy

$$\hat{x}_i(t) = \begin{cases} x_i^h(t), & \text{if } t_0 - t_h \leq t \leq t_0, \\ f_i(\hat{x}_1(t-\tau_{i1}), \hat{x}_2(t-\tau_{i2}), \ldots, \hat{x}_n(t-\tau_{in})), & \text{if } t_0 \leq t \leq t_E, \end{cases} \tag{6}$$

for $1 \leq i \leq n$. The following arguments are required for `bdesolve`:

`bdefun`	A handle to a function specifying the forms of the f_is (logic gates) in (1).
`lags`	The delay parameters. These can be inputted as a vector or as a matrix.
`history`	An initial history for the solution, inputted as a structure or as a column vector. In the latter case, each variable is taken to have a constant value over the interval $(-\infty, t_0)$.
`tspan`	The timespan over which to solve the BDEs. This can also be a single value `tEnd`, which is taken as the end time, if the history is given as a structure. In such cases, the initial value for the timespan is assumed to be the last time point in the history.

The following arguments are optional:

`forcing`	A structure, with the same form as a solution, that spans the full length of the prediction (*i.e.* `forcing.x` must span the same range as `history.x` and `tspan` combined).
`options`	A structure which can specify the maximum number of switches and/or the maximum number of function evaluations permitted during the generation of a solution.

A solution generated by `bdesolve` is given in the form of a structure that contains two fields, `x` and `y`. The field `x` is a real-valued vector of switch points: these are the time points at which at least one variable changes value. The field `y` is a logical matrix in which each row represents a variable and each column gives the values of the variables after each switch point in `x`. The output of `bdesolve` is defined over the interval given by the timespan argument. Therefore, a time point is included for the final point in the timespan, regardless of whether or not a switch occurs there. Fig. 1(b) plots the solution to the

simple negative feedback model (3) computed using `bdesolve` for the history specified in (4), using the following MATLAB commands:

```
tau1 = 1;
tau2 = 1;
% Define the delays
lags = [tau1, tau2];
% Constant history for all time t<0
history = [false; true];
% Generate solution for 5.5 units of time
tspan = [0 5.5];
% The model equations, where Z contains the values of the
% memorisation variables.
fun = @(Z) [~Z(2,2); Z(1,1)];
nsol = bdesolve(fun, lags, history, tspan);
% Make the solution plot-ready
nsolPR = bdePR(nsol, 1.1);
plot(nsolPR.x, nsolPR.y, 'LineWidth', 2);
xlim([0 5.5]);
ylim([0 2.3]);
yticks([0.5 1.6]);
yticklabels({'A', 'B'});
xlabel('Time, t');
```

Fig. 1(c) plots the time course of the corresponding analytical solution (5), which has been generated in BDEtools using the MATLAB commands below:

```
% Specify the time points of switches
asol.x = [0 1 2 3 4 5 5.5];
% Specify the states after each switch in asol.x
asol.y = [[0;0], [1;0], [1;1], [0;1], [0;0], [1;0], [1;0]];
% Make the solution plot-ready
% (i.e. convert to straight line segments)
asolPR = bdePR(asol, 1.1);
plot(asolPR.x, asolPR.y, 'LineWidth', 2);
xlim([0 5.5]);
ylim([0 2.3]);
yticks([0.5 1.6]);
yticklabels({'A', 'B'});
xlabel('Time, t');
```

Comparing Fig. 1(b) with Fig. 1(c) shows that the numerical solution exactly matches the analytical solution in this case. We note that in the code used to obtain the numerical solution, the model equations are written in terms of a matrix Z which is calculated by `bdesolve` and contains the memorisation variables:

$$Z = \begin{pmatrix} A(t-\tau_1) & A(t-\tau_2) \\ B(t-\tau_1) & B(t-\tau_2) \end{pmatrix}.$$

The pseudocode for `bdesolve` is given in Algorithm 1, and implements the numerical solution method developed by Dee & Ghil (Dee and Ghil, 1984). If a switch occurs at a particular point in time, it must be because a memorisation variable switches at that point. Therefore, given any switch, we can determine the candidates for switches at future times, although whether or not a switch occurs in the future depends on the values of the memorisation variables at that time. The solver works by calculating the times of candidate switch points in the future based on the times of already-calculated switches, or switches in the history. Then, at each candidate point in the future, the values of the memorisation variables are

evaluated and these serve as the input to the user-defined function that contains the model equations. The current state of each of the variables is calculated and if a switch occurs, it is added to the solution and candidates for switch points arising from it are computed. If not, it is discarded and the next candidate switch point is assessed.

Algorithm 1: Pseudocode for `bdesolve`. This is the basic version of the solver, *i.e.* without any optional arguments, such as the inclusion of forcing.

```
Require: bdefun                                              // The model equations
Require: lags                                                // The delay values
Require: history                                  // Initial history preceding the prediction
Require: tspan                                    // Timespan over which to calculate the solution
 1: cand_switches ← history.x + lags
      // Calculate candidate switches from history
 2: xNow = tspan(1)
      // Define first time point
 3: while xNow <= tspan(end) do
 4:    mem_times ← xNow - lags
      // Calculate times in past for each delay
 5:    mem_vars ← sol.y(mem_times)
      // Calculate memorisation variables at current timepoint
 6:    yNow ← bdefun(mem_vars)
      // Calculate values of variables at this point
 7:    if sol.y(end) != yNow then
 8:       sol.y = [sol.y, yNow]
          // If a switch occurs, append it to the solution
 9:       sol.x = [sol.x, xNow]
10:       cand_switches = [cand_switches, xNow + lags]
          // Update the list of candidate switches
11:    end if
12:    xNow ← min{a ∈ cand_switches | a ≥ xNow}
      // Move on to the next candidate switch point
13: end while
14: if sol.x(end) != tspan(end) then
15:    sol.y(end) = [sol.y, sol.y(end)]
      // Include the final timepoint in the solution
16:    sol.x(end) = [sol.x, tspan(end)]
17: end if
18: return sol
```

There are some details that are not mentioned in Algorithm 1 for simplicity, but which we feel are worth discussing. Firstly, our implementation of line 4 is not as simple as directly subtracting each delay from `xNow`, as any round-off error can affect the calculation of the memorisation variables. Since we are trying to determine the exact time at which a switch occurs, the value of `y` that we calculate from this can be very sensitive to such error and we may erroneously miss a switch point, resulting in an incorrectly calculated solution. To avoid this, we also include a vector to record the indices of which variables are the source of switches in `cand_switches`. This is then referenced when calculating the memorisation variable values to ensure that the calculation is correct.

We now expand upon our implementation of lines 1 and 10 in Algorithm 1. These can be implemented by `bdesolve` in one of two ways, depending on the form of the `lags` argument that is passed to it. If the user chooses to pass `lags` as a vector, it is not specified which delay is associated with which variable

and at each iteration the solver adds every delay to `xNow` in lines 1 and 10. Consequently, the solver performs more iterations of the `while` loop than are strictly necessary. Although specifying `lags` as a vector can lead to an inefficient computation, it is easy for the user to implement and it may make a negligible difference to the total CPU time used from a practical point of view. If, however, the user is concerned with computation time – for example if they expect a large number of switches or if they are calculating a large set of model predictions – there is a second option available. In this case, the user can provide `lags` as a matrix in which each non-zero element is a delay parameter, the number of rows equals the number of variables and the number of columns equals the number of delays. The solver utilises this extra information and only needs to add the delay values associated with any variables that have switched at `xNow`.

3.2. BDEtools utility functions

The BDEtools package also contains a number of other useful functions. For example, given discretisation thresholds $\{D_1, \ldots, D_n\}$, `bdediscrete` converts a set of real-valued timeseries $\mathbf{X}(t) = (X_1(t), \ldots, X_n(t))$ into a set of Boolean timeseries $\mathbf{x}(t) = (x_1(t), \ldots, x_n(t))$, through the following transformation

$$x_i(t) = \begin{cases} 0, & \text{if } \min_t X_i(t) \leq X_i(t) \leq D_i \left(\max_t X_i(t) - \min_t X_i(t)\right), \\ 1, & \text{otherwise,} \end{cases} \tag{7}$$

for $1 \leq i \leq n$.

The function `bdedist` calculates the Hamming distance between two Boolean time series $\mathbf{x}(t) = (x_1(t), \ldots, x_n(t))$ and $\mathbf{y}(t) = (y_1(t), \ldots, y_n(t))$ defined over the same time interval $t_1 \leq t \leq t_2$:

$$H(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} \int_{t_1}^{t_2} |x_i(t) - y_i(t)| \, dt. \tag{8}$$

The Hamming distance can be used to assess the accuracy of a numerical solution to a BDE system, as discussed in the next section, and also to quantify the goodness-of-fit of a BDE solution to a target dataset.

A list of the key functions in BDEtools is given in Table 1. Help for each of these functions, containing a description of arguments and output as well as some examples of use, can be obtained by typing `help function_name` at the MATLAB command prompt.

3.3. The serial BDE solver: `bdesolveserial`

Included in BDEtools is another solver, `bdesolveserial`, in which each variable is computed independently of the others from a precalculated timeseries, assumed to be a dataset that the model of interest is attempting to reproduce. Accordingly, we refer to this calculation method as solving in *series*, in contrast to the method implemented by `bdesolve`, which we refer to as solving in *parallel*. Writing the data as

$$\mathbf{x}_D(t) = \left(x_1^D(t), \ldots, x_n^D(t)\right) : t_0 \leq t \leq t_E,$$

where $t_E \geq t_0 + \max(\boldsymbol{\tau})$, `bdesolveserial` generates a set of Boolean timeseries $\{\hat{\mathbf{x}}(t) = (\hat{x}_1(t), \ldots, \hat{x}_n(t)) : t_0 \leq t \leq t_E\}$ that satisfy

$$\hat{x}_i(t) = \begin{cases} x_i^D(t), & \text{if } t_0 \leq t \leq t_0 + \max(\boldsymbol{\tau}), \\ f_i\left(x_1^D(t - \tau_{i1}), x_2^D(t - \tau_{i2}), \ldots, x_n^D(t - \tau_{in})\right), & \text{if } t_0 + \max(\boldsymbol{\tau}) \leq t \leq t_E, \end{cases} \tag{9}$$

TABLE 1: List of the primary BDEtools functions, with a brief description of each.

Function name	Description
`bdecut`	Cuts a solution at a given time point into two separate solutions.
`bdediscrete`	Converts real-valued timeseries data to a set of switches.
`bdedist`	Computes the Hamming distance between two Boolean timeseries.
`bdejoin`	Splices together two BDE solutions that are defined over adjoining time intervals.
`bdemerge`	Merges two BDE solutions that are defined over the same time range.
`bdeopts`	Defines some options for the parallel BDE solver.
`bdePR`	Makes a solution plot-ready by converting it to a set of lines.
`bdeplot`	Plots a BDE solution as offset timeseries.
`bdesep`	Extracts variables from a BDE solution.
`bdesolve`	The parallel BDE solver.
`bdesolveserial`	The serial BDE solver.
`bdeval`	Returns the values of a BDE solution at specified time points.
`default_bdeopts`	The default options for the parallel BDE solver.

for $1 \leq i \leq n$. The function `bdesolveserial` is called in much the same way as `bdesolve`, but the precalculated timeseries – rather than a history – is passed to the function as an argument. The timeseries should be compatible with the model equations, *i.e.* it should have the same number of variables and be defined over the same range as the required solution. The pseudocode for `bdesolveserial` is given in Algorithm 2. The major differences from the `bdesolve` pseudocode are in line 1, reflecting the fact that the initial set of candidate switches is calculated from the inputted data rather than a history, and line 5, reflecting the fact that the memorisation variables are now computed directly from the data (*cf.* eqns. (6) and (9)).

Solving a system of BDEs in series can be useful, for a number of reasons. Firstly, consider an *exact* prediction obtained using the parallel solver with the data as an initial history, *i.e.* a timeseries $\hat{\mathbf{x}}(t)$ generated using `bdesolve` from the history

$$\mathbf{x}_h(t) = \left(x_1^D(t), \ldots, x_n^D(t)\right) : t_0 \leq t \leq t_0 + \max(\boldsymbol{\tau}),$$

such that $\{\hat{\mathbf{x}}(t) = \mathbf{x}_D(t) : t_0 \leq t \leq t_E\}$. Then clearly, this prediction will also satisfy the serial solver equations (9). However, the converse is not true: an exact prediction $\hat{\mathbf{x}}(t)$ generated using the serial solver from the data $\mathbf{x}_D(t)$ will not necessarily solve the parallel equations (6). Accurate predictions obtained using the serial solver thus generate timeseries that are *consistent* with the data for the given model and some of these will also be true solutions of the BDE, *i.e.* can be generated using the parallel solver. Serial updating can thus be considered as a computationally cheap means of generating putative solutions that match the data (Akman *et al.*, 2012; Akman and Fieldsend, 2020).

In addition, the serial solver can be used to quantify the extent to which a Boolean timeseries $\hat{\mathbf{x}}(t)$ generated using `bdesolve` satisfies the BDE system (1), by using $\hat{\mathbf{x}}(t)$ as the input to `bdesolveserial` in order to check the consistency of the timeseries with the equations. Writing the resulting output of `bdesolveserial` as $\hat{\mathbf{x}}_S(t)$, the Hamming distance $H(\hat{\mathbf{x}}, \hat{\mathbf{x}}_S)$ can then be taken as a measurement of the accuracy of the solution generated by the parallel solver, with an H value of zero indicting an exact solution. As an example of this, the following MATLAB commands compute $H(\hat{\mathbf{x}}, \hat{\mathbf{x}}_S)$ for the solution

Algorithm 2: Pseudocode for `bdesolveserial`. This is the basic version of the solver, *i.e.* without any optional arguments, such as the inclusion of forcing.

Require: `bdefun` // The model equations
Require: `lags` // The delay values
Require: `data` // The data from which the prediction is generated
Require: `tspan` // Timespan over which to calculate the solution

`cand_switches` ← `data.x + lags`
  `// Calculate candidate switches from data`
`xNow = tspan(1)`
  `// Define first time point`
while `xNow <= tspan(end)` **do**
  `mem_times` ← `xNow - lags`
  `// Calculate times in past for each delay`
  `mem_vars` ← `data.y(mem_times)`
  `// Calculate memorisation variables at current timepoint`
  `yNow` ← `bdefun(mem_vars)`
  `// Calculate values of variables at this point`
  if `sol.y(end) != yNow` **then**
    `sol.y = [sol.y, yNow]`
    `// If a switch occurs, append it to the solution`
    `sol.x = [sol.x, xNow]`
    `cand_switches = [cand_switches, xNow + lags]`
    `// Update the list of candidate switches`
  end if
  `xNow` ← $min\{a \in$ `cand_switches` $\mid a \geq$ `xNow`$\}$
  `// Move on to the next candidate switch point`
end while
if `sol.x(end) != tspan(end)` **then**
  `sol.y(end) = [sol.y, sol.y(end)]`
  `// Include the final timepoint in the solution`
  `sol.x(end) = [sol.x, tspan(end)]`
end if
return `sol`

to the simple negative feedback model (3) generated with the history defined in (4), which was used to illustrate the application of `bdesolve` in Fig. 1:

```
tau1 = 1;
tau2 = 1;
% Define the delays
lags = [tau1, tau2];
% Constant history for all time t<0
history = [false; true];
% Generate solution for 5.5 units of time
tspan = [0 5.5];
% The model equations, where Z contains the values of the
% memorisation variables.
fun = @(Z) [~Z(2,2); Z(1,1)];
% Solve with the parallel solver.
xh = bdesolve(fun, lags, history, tspan);
% Feed parallel solution into the serial solver.
xhs = bdesolveserial(fun, lags, xh, [tspan(1)+max(lags)  tspan(end)]);
% Append the segment of the parallel solution used for projection to the serial solution.
xhs.x(1) = [];
xhs.y(:,1) = [];
```

```
[xhcut,~] = bdecut(xh, xhs.x(1));
xhs = bdejoin(xhcut, xhs);
% Calculate the Hamming distance between the parallel and serial solutions.
d = bdedist(xh, xhs);
H = sum(d)
>> H =

           0
```

As can be seen from the above, $H(\hat{\mathbf{x}}, \hat{\mathbf{x}}_S)$ is calculated to be zero, confirming that `bdesolve` has generated an exact solution of eqns. (3) in this case. Note that such a simple measure of solution accuracy is not typically available with other modelling approaches, such as ODEs.

4. RESULTS

4.1. Case study: A circadian oscillator

Here, we present one of the examples included with BDEtools to demonstrate its functionality: `circadian_example`, which simulates a minimal model of circadian oscillations in the fungus *Neurospora crassa*. Viewing the MATLAB code for the example can be instructive for learning how the various functions in BDEtools can be used (see the Appendix). `circadian_example` is based on some previous work of ours (Akman *et al.*, 2012; Doherty *et al.*, 2017; Akman and Fieldsend, 2020) and shows how real-valued experimental data can be converted to Boolean data, and then used as an initial history for calculating a solution to a system of BDEs, thereby enabling the model prediction to be compared directly to experimental measurements. The example also demonstrates how: (i) a forcing input can be provided to the solver; and (ii) a BDE system can be solved in series as well as in parallel. The model comprises a simple set of equations representing a gene-protein negative feedback loop that is capable of replicating the typical hallmarks of a circadian system – autonomous oscillations with a circadian period that can be entrained by a light-dark cycle with a 24h period. The model equations are

$$\begin{aligned} F_M(t) &= \neg F_P(t-\tau_2) \vee L(t-\tau_3), \\ F_P(t) &= F_M(t-\tau_1), \end{aligned} \tag{10}$$

where the variables $F_M(t)$ and $F_P(t)$ represent whether or not the mRNA or protein product, respectively, of the *Neurospora FREQUENCY* (*FRQ*) gene is above a certain threshold at time t. Negative feedback is modelled using the NOT ($\neg$) operator: FRQ protein represses the production of *FRQ* mRNA. The variable $L(t)$ describes the 24h periodic light forcing and is defined as

$$L(t) = \begin{cases} 1 \text{ if } 6 < \operatorname{mod}(t, 24) \leq 18, \\ 0 \text{ otherwise}, \end{cases}$$

modelling a light signal that switches on when $t = 6$ (dawn) and switches off when $t = 18$ (dusk). The positive effect of light on *FRQ* transcription is modelled using the OR ($\vee$) operator.

The output from `circadian_example` is shown in Fig. 2. Firstly, some synthetic mRNA and protein timeseries are generated from the ODE formulation of the model (Leloup *et al.*, 1999) using the numerical procedure described in detail in (Akman *et al.*, 2012). Data for model-fitting is then obtained by sampling these timeseries every 0.5h (Fig. 2(a)). Next, the function `bdediscrete` is used to convert the real-valued data to a BDE solution, using discretisation thresholds that are arbitrarily taken to be the minimum values

of the variables plus 0.3 times their peak-to-peak amplitudes (Fig. 2(b) – *cf.* (7)). Where a threshold value lies between two successive data points, the switch point is calculated as the time at which the linear interpolant joining the two data points crosses the threshold.

Fig. 2(c) plots the Boolean timeseries $\hat{\mathbf{x}}_S(t)$ generated by solving eqns. (10) in serial with $\mathbf{x}_D(t)$ as the input, with delay values $\{\tau_1, \tau_2, \tau_3\}$ set equal to the average values of the times between switches in the data (*e.g.* τ_1 is set equal to the average time between switches in $F_M(t)$ and $F_P(t)$). By comparing $\hat{\mathbf{x}}_S(t)$ with $\mathbf{x}_D(t)$, it can be seen that for this choice of delays, the prediction obtained with `bdesolveserial` is quite close to the target dataset. This can be quantified by a *cost function* $C\left(\hat{\mathbf{x}}_S, \mathbf{x}_D\right)$, which measures the discrepancy between $\hat{\mathbf{x}}_S(t)$ and $\mathbf{x}_D(t)$ as the average Hamming distance across different variables, normalised by the data timespan, that is

$$C\left(\hat{\mathbf{x}}_S, \mathbf{x}_D\right) = \frac{1}{t_E - t_0} \frac{1}{2} H\left(\hat{\mathbf{x}}_S, \mathbf{x}_D\right),$$

where $H(\cdot, \cdot)$ was defined in (8) (Akman *et al.*, 2012; Doherty *et al.*, 2017; Akman and Fieldsend, 2020; Alyahya *et al.*, 2021). $C\left(\hat{\mathbf{x}}_S, \mathbf{x}_D\right)$ is calculated to be 0.08, indicating a 92% agreement between the prediction and the data, verifying that the data are consistent with the model equations for the chosen delays. The good agreement suggests that the chosen delay combination is likely to yield a parallel solution to the equations that also matches the data, *i.e.* a true solution of (10) generated using an initial data cycle as the history, for which the projection beyond the history yields a good prediction. Fig. 2(d) shows that this is indeed the case – the parallel solution $\hat{\mathbf{x}}$ generated by `bdesolve` using the first 24h of $\mathbf{x}_D(t)$ as the initial history can be seen to match the data closely. In fact, the discrepancy $C\left(\hat{\mathbf{x}}, \mathbf{x}_D\right)$ between the parallel prediction and the data is also 0.08 in this case.

5. DISCUSSION

BDEs provide a method for fitting timeseries data that reduces the complexity of the model construction and model calibration procedures, in comparison to more commonly used simulation methods, such as those based on differential equations (Ghil *et al.*, 2008; Akman *et al.*, 2012). In particular, BDE systems contain significantly fewer parameters, thereby substantially reducing the search space for optimisation. For example, when modelling GRNs, all the parameters controlling the delays associated with the transcription of an mRNA from a gene, its translation into protein and its action as a transcription factor on a downstream target are telescoped into a single delay parameter (Akman *et al.*, 2012). This compression of the parameter space can help mitigate issues related to parameter identification and overfitting that are common when using differential equation-based methods, as well as reducing the computation time required to obtain solutions. Furthermore, there is a natural cost function for fitting BDE systems to data, based on the Hamming distance between bitstrings. This is not generally the case when fitting continuous data, where there are a number of possible cost functions (*e.g.* least-squares), each of which defines a different optimisation problem (Locke *et al.*, 2005; Akman *et al.*, 2010; Adams *et al.*, 2013).

The BDE modelling framework also allows for the enumeration of alternative model architectures, enabling the structure of the model itself to be parametrised by logical variables. For example, in the case where the activity of a variable x_i is affected by two input variables, the logical AND operator represents the case where both inputs need to be on to elicit a response, whilst the logical OR operator represents the case where either input alone can switch on the output variable. The logic gate f_i that determines this input-output relationship can thus be parametrised by a single Boolean parameter. More complex input-output relationships can be represented by longer bitstrings. Gathering together the

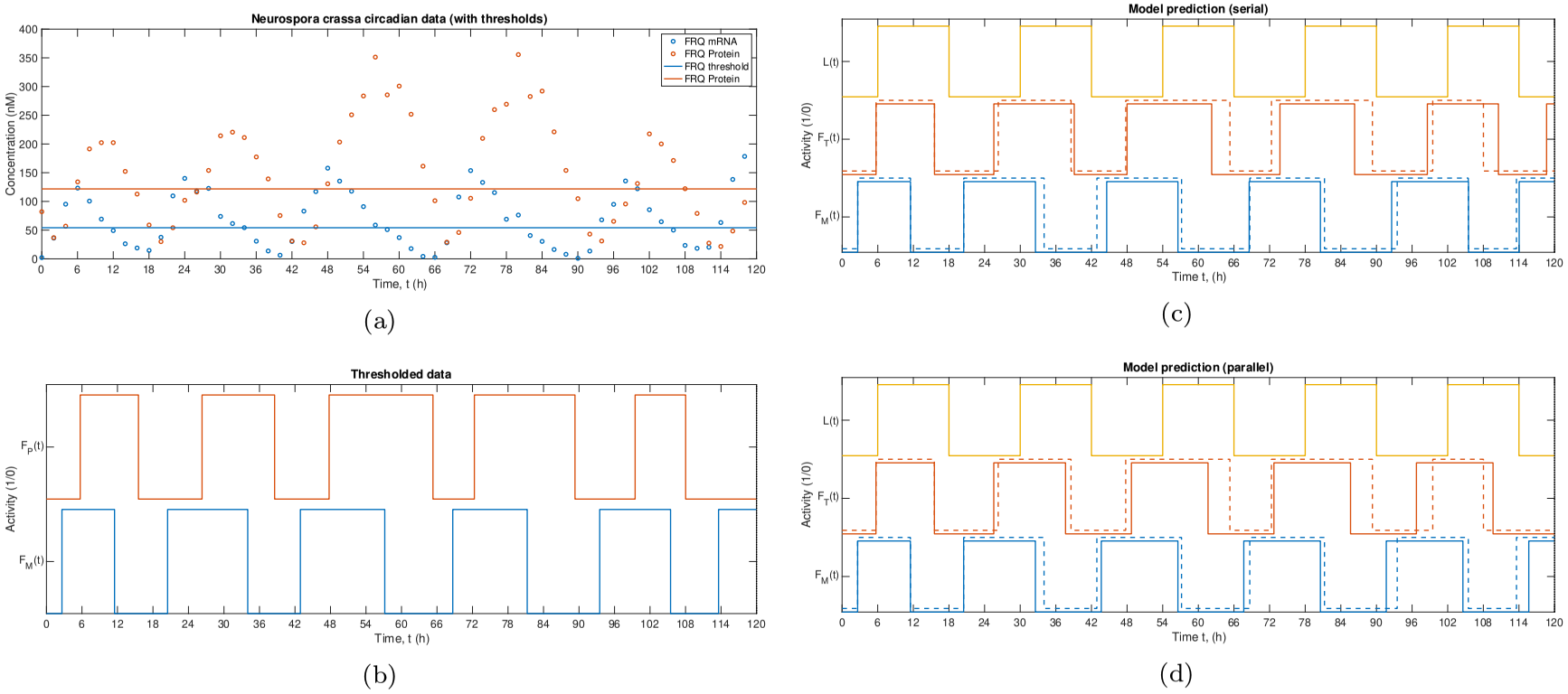


FIG. 2: The output of `circadian_example`. **(a)** Synthetic continuous mRNA and protein data used to generate model predictions. **(b)** Boolean data $\mathbf{x}_D(t)$ obtained by thresholding the continuous data using the `bdediscrete` function, with the thresholds $D_1 = 0.3$ and $D_2 = 0.3$ (these thresholds are plotted as horizontal lines in **(a)**). **(c)** The solution $\hat{\mathbf{x}}_S(t)$ of eqns. (10) generated from the Boolean data using the serial solver `bdesolveserial` with $\{\tau_1 = 5.0752, \tau_2 = 6.0211, \tau_3 = 14.5586\}$. **(d)** The solution $\hat{\mathbf{x}}(t)$ generated using the parallel solver `bdesolve`, with the first 24h of $\mathbf{x}_D(t)$ as the initial history.

bitstrings for each f_i thus yields a set of Boolean metaparameters that encode all the possible models consistent with a given directed graph specifying the connections between model components. This leads to mixed-integer optimisation problems in which model architectures and parameter values are fitted to data simultaneously, thereby enabling BDE models to be used for network inference as well as parameter-fitting (Sevim *et al.*, 2010; Akman *et al.*, 2012; Wynn *et al.*, 2012; Doherty *et al.*, 2017; Akman and Fieldsend, 2020).

In this paper, we have presented BDEtools, an open-source MATLAB package that enables BDE models to be written and simulated in a relatively straightforward fashion. Furthermore, to demonstrate the functionality of the package, we have provided an example of its application to an established circadian biology model. We have described the main (parallel) BDE solver algorithm, as well as some of the other useful package routines, such as code for discretising continuous data and for calculating the Hamming distance between Boolean timeseries. In addition to the parallel solver, we have described the serial solver, which generates the prediction for each BDE variable directly from a given set of timeseries data (*i.e.* independently of the other variables). This solver can be used as a computationally inexpensive way of determining suitable parameter values for a model, and was utilised in (Akman *et al.*, 2012) to fit BDE circadian clock models to continuous data – some of these optimal fits are included as unit tests in BDEtools (see Fig. 3(a) for an example). The serial solver also provides a simple means of assessing the accuracy of a solution to a set of BDEs generated by the parallel solver (Fig. 3(b)).

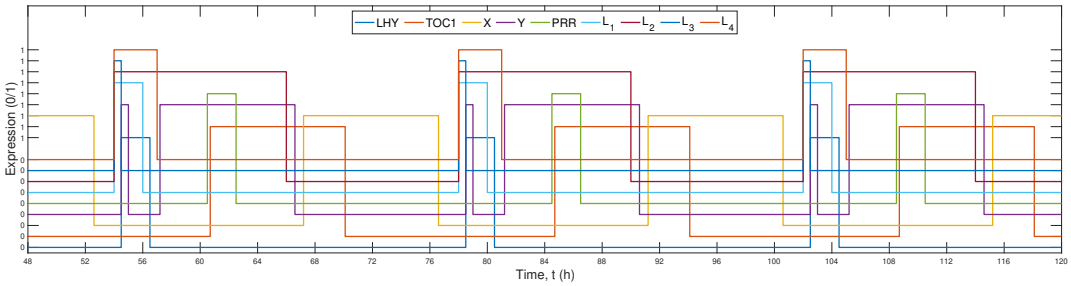
We anticipate that BDEtools will encourage more researchers to exploit the power of BDE models, particularly the extent to which they simplify the parameter optimisation problem. As part of our work in this area, we have begun developing a suite of evolutionary optimisers specifically adapted to BDE systems that can be used together with a basic Python implementation of the parallel solver (Fieldsend *et al.*, 2021). This suite currently contains a particle swarm optimiser that can identify multiple local optima within a given cost function landscape (Fieldsend, 2014) and an algorithm that utilises elite accumulative sampling to locate optima that are robust to disturbances in the parameter space (Alyahya *et al.*, 2017).

Finally, in terms of future development of the package, we are aiming to expand the scope of BDEtools by modifying the solver algorithms to include the following:

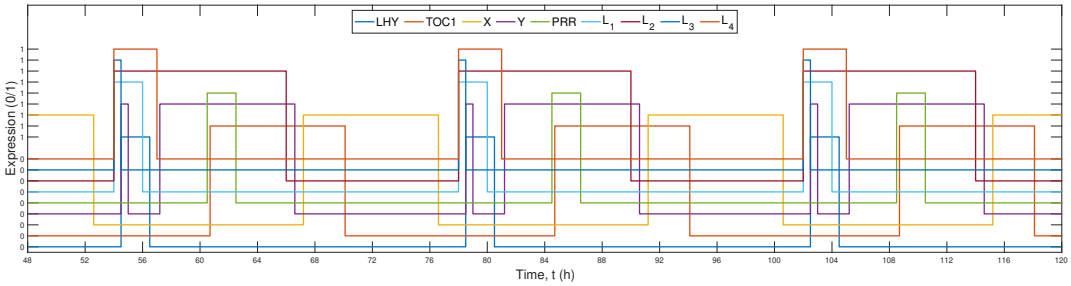
- Short pulse rejection, obtained by imposing a refractory period within which no jumps in the BDE variables are possible (Öktem *et al.*, 2003; Zhang *et al.*, 2009). Whilst it has been established for some time that BDEs can generate periodic and quasiperiodic oscillations (Saunders and Ghil, 2001), short pulse rejection increases the repertoire of BDE dynamics to include chaotic oscillations (Zhang *et al.*, 2009).

- Asymmetric delays, in which each τ_{ij} in (1) is split into a delay τ_{ij+} controlling $1\rightarrow0$ transitions and a separate delay τ_{ij-} controlling $0\rightarrow1$ transitions (Thomas, 1973, 1978; Öktem *et al.*, 2003; Sevim *et al.*, 2010). This extension to the basic BDE structure could be implemented by incorporating a routine that automatically parses the MATLAB function containing the BDE equations to generate the asymmetric logic functions required (Öktem *et al.*, 2003), and by modifying the input solver parameter `lags` to take two values for each delay.

- Stochastic delays, in which the τ_{ij}s are drawn from a probability distribution (Klemm and Bornholdt, 2005). The inclusion of stochasticity would require the list of candidate switches in Algorithms 1 and 2 to be randomised in such a way that causality is still preserved (Klemm and Bornholdt, 2005). Probabilistic switching could further broaden the scope of BDE modelling – for example



(a)



(b)

FIG. 3: (a) A solution $\hat{\mathbf{x}}(t)$ to the BDE formulation of the *Arabidopsis* circadian clock model introduced in (Locke *et al.*, 2006), generated using `bdesolve`. The solution is simulated in a 12L:12D light-dark cycle with dawn at $t = 6$h and dusk at $t = 18$h, using the optimal parameter values from (Akman *et al.*, 2012). The model has nine variables: five clock genes $\{LHY(t), TOC1(t), X(t), Y(t), PRR(t)\}$ and four forcing light inputs $\{L_1(t), L_2(t), L_3(t), L_4(t)\}$. **(b)** The solution $\hat{\mathbf{x}}_S(t)$ generated using the serial solver `bdesolveserial` with $\hat{\mathbf{x}}(t)$ as the inputted timeseries. $H(\hat{\mathbf{x}}, \hat{\mathbf{x}}_S)$ is zero, indicating that $\hat{\mathbf{x}}(t)$ is an accurate solution of the corresponding BDE equations in this case. The plots are generated as part of the unit test routine `arabid3lp_test_wplot`.

by enabling BDE versions of stochastic circadian clock models to be constructed (Guerriero *et al.*, 2014).

6. CONCLUSIONS

BDE models possess the capacity to simulate complex dynamics, despite their reduced complexity and parametrisation. We have developed the BDEtools package to provide researchers with software for developing biological network models constructed on the BDE formalism. We intend to continue adding new BDE systems to the current suite of models available in BDEtools, with the aim of developing a curated database similar to the BioModels project (Li *et al.*, 2010). Indeed, we hope that this database – together with the other BDEtools functions – will provide the computational biology community with a useful resource with which to explore the utility of BDE modelling.

7. APPENDIX

circadian_example – MATLAB code

```
function circadian_example

% Specify final timepoint of prediction.

tEnd = 120;

% Define the model parameters (delays). The following are the average times between
% switches in the data (these have been calculated in advance).

tau1 = 5.0752;
tau2 = 6.0211;
tau3 = 14.5586;
lags = [tau1, tau2, tau3];

% Load data to be used for the history.

load('neur_circ_data_gillespie.mat');

% Load the synthetic dataset generated using the Gillespie algorithm.

xData = neur_circ_data_gillespie.LD(3, :);
yData = neur_circ_data_gillespie.LD(1:2, :);

% Set thresholds for discretising the data.

T = [0.3 0.3];

% Convert real-valued data to Boolean data by thresholding.

synData = bdediscrete(xData, yData, T);

% Add an extra point at the end to give the data the same tRange as the prediction.

if synData.x<tEnd
    synData.x = [synData.x, tEnd];
    synData.y = [synData.y, synData.y(:, end)];
end

% Specify the history.

history.x = synData.x(synData.x < 24); % Define history as all points from t=0 to 24.
history.y = synData.y(:, synData.x < 24);

% Augment the history. Include a point at t=24 to make the history a full
% 24 hours (note: only the values from 24 - max(lags) to 24 will be used in
% calculating the solution).

history.x = [history.x, 24];
history.y = [history.y, history.y(:, end)];

% Specify the forcing.

forcing.x = [0, 6, 18:12:114, 120];
forcing.y = mod(forcing.x, 24) >= 6 & (mod(forcing.x, 24) < 18);

% Plot the data.

plot_neurospora_data(xData, yData, T, synData.x, synData.y);
```

```
59
60  % Solve the equations in parallel.
61
62  solPar = bdesolve(@neurospora_eqns, lags, history, tEnd, forcing);
63
64  % Solve the equations in serial.
65
66  solDataSer = bdesolveserial(@neurospora_eqns, lags, synData, [24 tEnd], forcing);
67
68  % Append the histories.
69
70  solDataSer_wHist = bdejoin(solDataSer.history, solDataSer);
71  solPar_wHist = bdejoin(solPar.history, solPar);
72
73  % Plot the serial solution.
74
75  figure;
76  plot_neurospora_prediction(solDataSer);
77  title('Model prediction (serial)');
78
79  % Add the data.
80
81  my_colours = get(gca,'colororder');
82  synDataPR = bdePR(synData, 1.1,0.05);
83  plot(synDataPR.x, synDataPR.y(1,:), '--', 'Color', my_colours(1, :), 'LineWidth', 2);
84  plot(synDataPR.x, synDataPR.y(2,:), '--', 'Color', my_colours(2, :), 'LineWidth', 2);
85
86  % Plot the parallel solution.
87
88  figure;
89  plot_neurospora_prediction(solPar);
90  title('Model prediction (parallel)');
91
92  % Add the data.
93
94  synDataPR = bdePR(synData, 1.1,0.05);
95  plot(synDataPR.x, synDataPR.y(1,:), '--', 'Color', my_colours(1, :), 'LineWidth', 2);
96  plot(synDataPR.x, synDataPR.y(2,:), '--', 'Color', my_colours(2, :), 'LineWidth', 2);
97
98  % Calculate the prediction errors.
99
100 [~, ~, C1] = bdedist(solDataSer_wHist, synData);
101 [~, ~, C2] = bdedist(solDataSer_wHist, synData);
102 disp(strcat('Prediction error, serial: ',num2str(C1)));
103 disp(strcat('Prediction error, parallel: ',num2str(C2)));
104
105 %------------------------------------------ %
106 %------------ SUBFUNCTIONS---------------- %
107 %------------------------------------------ %
108
109 % Subfunction that defines the model.
110
111 function X = neurospora_eqns(Z1, Z2)
112
113 M = ~Z1(2, 2) | Z2(1, 3);
114 FT = Z1(1, 1);
115
116 X = [M; FT];
117
118 % Subfunction for plotting the continuous and discretised data.
119
120 function plot_neurospora_data(xData, yData, T, xDataBool, yDataBool)
121
122 offset = 1.1;
123 data.x = xDataBool;
```

```
data.y = yDataBool;
dataPR = bdePR(data, offset);

T1 = min(yData(1, :)) + T(1) * (max(yData(1, :)) - min(yData(1, :)));
T2 = min(yData(2, :)) + T(2) * (max(yData(2, :)) - min(yData(2, :)));

figure;
my_colours = get(gca,'colororder');
plot(xData, yData, 'Marker', 'o', 'LineStyle', 'none', 'LineWidth', 2); % Plot the data.
hold on ;
plot([0 120], [T1, T1], 'Color', my_colours(1, :), 'LineWidth', 2);
plot([0 120], [T2, T2], 'Color', my_colours(2, :), 'LineWidth', 2);
legend('FRQ mRNA', 'FRQ Protein','FRQ threshold','FRQ Protein');
xlabel('Time, t (h)');
ylabel('Concentration (nM)');
title('Neurospora crassa circadian data (with thresholds)');
set(gca,'Xtick',0:6:120);
set(gca,'Fontsize',14);

figure;

% Plot discretised data.

plot(dataPR.x, dataPR.y, 'LineWidth', 2)
hold on;

yticks([0.5, 1.6]); % Have one ytick for each variable.
yticklabels({'F_M(t)', 'F_P(t)'}); % Name each ytick.
ylim([0 2.2]);
xlabel('Time, t (h)');
ylabel('Activity (1/0)');
title('Thresholded data');
set(gca,'Xtick',0:6:120);
set(gca,'Fontsize',14);

% Subfunction for plotting model predictions.

function plot_neurospora_prediction(sol)

solPlot = bdejoin(sol.history, sol);
solPlot = bdemerge(solPlot, sol.forcing);

offset = 1.1;
tEndHist = sol.history.x(end);
solPR = bdePR(solPlot, offset);

plot(solPR.x, solPR.y, 'LineWidth', 2);
hold on;
yticks([0.5, 1.6, 2.7]);
yticklabels({'F_M(t)','F_T(t)','L(t)'});
ylim([0 3.3]);
xlabel('Time t, (h)');
ylabel('Activity (1/0)');
set(gca,'Xtick',0:6:120);
set(gca,'Fontsize',14);
```

AUTHORS' CONTRIBUTIONS

OEA designed the study, OEA, KD and BJW developed software, OEA and KD wrote the article. All authors read and approved the article.

ACKNOWLEDGEMENTS

We would like to thank Jonathan Fieldsend and Khulood Alyahya for useful discussions. The authors would also like to acknowledge the use of the University of Exeter High-Performance Computing (HPC) facility in carrying out this work.

DATA AVAILABILITY

The BDEtools package is under version control with `git` at GitHub: https://github.com/oeakman/BDEtools. BDEtools is released under the MIT license and requires MATLAB R2017a or later.

AUTHOR DISCLOSURE STATEMENT

The authors declare that they have no competing financial interests.

FUNDING INFORMATION

This work was financially supported by the Engineering and Physical Sciences Research Council (grant nos. EP/K040987/1 and EP/N017846/1).

REFERENCES

Adams, R., Clark, A., Yamaguchi, A., Hanlon, N., Tsorman, N., Ali, S., Lebedeva, G., Goltsov, A., Sorokin, A.A., Akman, O.E., Troein, C., Millar, A.J., Goryanin, I., and Gilmore, S., 2013. SBSI: an extensible distributed software infrastructure for parameter estimation in systems biology. *Bioinformatics* 29, 664–665.

Akman, O.E. and Fieldsend, J.E., 2020. Multi-objective optimisation of gene regulatory networks: Insights from a Boolean circadian clock model. In *Proc. BICOB 2020*, volume 70, 149–162.

Akman, O.E., Watterson, S., Parton, A., Binns, N., Millar, A.J., and Ghazal, P., 2012. Digital clocks: simple Boolean models can quantitatively describe circadian systems. *J. Roy. Soc. Interface* 9, 2365–2382.

Akman, O. E., Rand, D. A., Brown, P. E., and Millar, A. J., 2010. Robustness from flexibility in the fungal circadian clock. *BMC Syst. Biol.* 4, 88.

Alyahya, K., Doherty, K., Akman, O.E., and Fieldsend, J.E., 2017. On the exploitation of search history and accumulative sampling in robust optimisation. In *Proc. GECCO 2017*, 185–186.

Alyahya, K., Doherty, K., Akman, O.E., and Fieldsend, J.E., 2021. Reduced models of gene regulatory networks: Visualising multi-modal landscapes. In Preuss, M., Epitropakis, M.G., Li, X., and Fieldsend, J.E., eds., *Metaheuristics for Finding Multiple Solutions*, 229–258. Springer International Publishing.

Berg, M., Plöntzke, J., Siebert, H., and Röblitz, S., 2021. Modelling oscillatory patterns in the bovine estrous cycle with Boolean delay equations. *Bull. Math. Biol.* 83, 121.

Coluzzi, B., Ghil, M., Hallegatte, S., and Weisbuch, G., 2011. Boolean delay equations on networks in economics and the geosciences. *Int. J. Bifurcat. Chaos* 21, 3511–3548.

Dee, D. and Ghil, M., 1984. Boolean difference equations, i: Formulation and dynamic behavior. *SIAM J. Appl. Math.* 44, 111–126.

Doherty, K., Alyahya, K., Akman, O.E., and Fieldsend, J.E., 2017. Optimisation and landscape analysis of computational biology models: A case study. In *Proc. GECCO 2017*, 1644–1651.

Fieldsend, J.E., 2014. Running up those hills: Multi-modal search with the niching migratory multi-swarm optimiser. In *Proc. CEC 2014*, 2593–2600.

Fieldsend, J.E., Akman, O.E., Alyahya, K., Doherty, K., Millar, A.J., Hume, A., Banglawala, N., Wood, C., Pitt, J., and Karatas, M.D., 2021. The parameter optimisation problem: Addressing a key challenge in computational systems biology. http://pop-project.ex.ac.uk.

Foo, M., Bates, D.G., and Akman, O.E., 2020. A simplified modelling framework facilitates more complex representations of plant circadian clocks. *PLoS Comput. Biol.* 16, e1007671.

Ghil, M. and Mullhaupt, A., 1985. Boolean delay equations. II. Periodic and Aperiodic Solutions. *J. Stat. Phys.* 41, 125–173.

Ghil, M., Mullhaupt, A., and Pestiaux, P., 1987. Deep water formation and quaternary glaciations. *Climate Dynamics* 2, 1–10.

Ghil, M., Zaliapin, I., and Coluzzi, B., 2008. Boolean delay equations: A simple way of looking at complex systems. *Physica D* 237, 2967–2986.

Guerriero, M.L., Akman, O.E., and van Ooijen, G., 2014. Stochastic models of cellular circadian rhythms in plants help to understand the impact of noise on robustness and clock structure. *Front. Plant Sci.* 5, 564.

Kauffman, S., Peterson, C., Samuelsson, B., and Troein, C., 2004. Genetic networks with canalyzing Boolean rules are always stable. *Proc. Natl. Acad. Sci. U.S.A.* 101, 17102–17107.

Klemm, K. and Bornholdt, S., 2005. Topology of biological networks and reliability of information processing. *Proc. Natl. Acad. Sci. U.S.A.* 102, 18414–18419.

Leloup, J.C., Gonze, D., and Goldbeter, A., 1999. Limit cycle models for circadian rhythms based on transcriptional regulation in *Drosophila* and *Neurospora*. *J. Biol. Rhythms* 14, 433–448.

Li, C., Donizelli, M., Rodriguez, N., Dharuri, H., Endler, L., Chelliah, V., Li, L., He, E., Henry, A., Stefan, M.I., Snoep, J.L., Hucka, M., Novere, N. Le, and Laibe, C., 2010. BioModels Database: An enhanced, curated and annotated resource for published quantitative kinetic models. *BMC Syst. Biol.* 4, 92.

Locke, J.C.W., Kozma-Bognar, L., Gould, P.D., Fehér, B., Kevei, E., Nagy, F., Turner, M.S., Hall, A., and Millar, A.J., 2006. Experimental validation of a predicted feedback loop in the multi-oscillator clock of *Arabidopsis thaliana*. *Mol. Syst. Biol.* 2, 59.

Locke, J. C. W., Millar, A. J., and Turner, M. S., 2005. Modelling genetic networks with noisy and varied experimental data: the circadian clock in Arabidopsis thaliana. *J. Theor. Biol.* 234, 383–93.

Montefusco, F., Akman, O.E., Soyer, O.S., and Bates, D.G., 2016. Ultrasensitive negative feedback control: A natural approach for the design of synthetic controllers. *PLoS ONE* 18, e0161605.

Nikolajewa, S., Friedel, M., and Wilhelm, T., 2007. Boolean networks with biologically relevant rules show ordered behavior. *Biosystems* 90, 40–47.

Öktem, H., Pearson, R., and Egiazarian, K., 2003. An adjustable aperiodic model class of genomic interactions using continuous time Boolean networks (Boolean delay equations). *Chaos* 13, 1167–1174.

Saunders, A. and Ghil, M., 2001. A Boolean delay equation model of ENSO variability. *Physica D* 160, 54–78.

Sevim, V., Gong, X., and Socolar, J.E.A., 2010. Reliability of transcriptional cycles and the yeast cell-cycle oscillator. *PLoS Comput. Biol.* 6, e1000842.

Steinacher, A., Bates, D. G., Akman, O. E., and Soyer, O. S., 2016. Nonlinear dynamics in gene regulation promote robustness and evolvability of gene expression levels. *PLoS One* 11, e0153295.

Thomas, R., 1973. Boolean formalization of genetic control circuits. *J. Theor. Biol.* 42, 563–585.

Thomas, R., 1978. Logical analysis of systems comprising feedback loops. *J. Theor. Biol.* 73, 631–656.

Thomas, R., 1991. Regulatory networks seen as asynchronous automata: A logical description. *J. Theor. Biol.* 153, 1–23.

Tokuda, I.T., Akman, O.E., and Locke, J.C.W., 2019. Reducing the complexity of mathematical models for the plant circadian clock by distributed delays. *J. Theor. Biol.* 463, 155–166.

Watterson, S. and Ghazal, P., 2010. Use of logic theory in understanding regulatory pathway signaling in response to infection. *Future Microbiol.* 5, 163–176.

Watterson, S., Marshall, S., and Ghazal, P., 2008. Logic models of pathway biology. *Drug Discov. Today* 13, 447–456.

Wright, D.G., Stocker, T.F., and Mysak, L.A., 1990. A note on quaternary climate modelling using Boolean delay equations. *Climate Dynamics* 4, 263–267.

Wynn, M.L., Consul, N., Merajver, S.D., and Schnell, S., 2012. Logic-based models in systems biology: a predictive and parameter-free network analysis method. *Integr. Biol.* 4, 1323–1337.

Yu, L., Watterson, S., Marshall, S., and Ghazal, P., 2008. Inferring Boolean networks with perturbation from sparse gene expression data: a general model applied to the Interferon regulatory network. *Mol. Biosyst.* 4, 1024–1030.

Zaliapin, I., Keilis-Borok, V., and Ghil, M., 2003a. A Boolean Delay Equation Model of Colliding Cascades. Part I: Multiple Seismic Regimes. *J. Stat. Phys.* 111, 815–837.

Zaliapin, I., Keilis-Borok, V., and Ghil, M., 2003b. A Boolean Delay Equation Model of Colliding Cascades. Part II: Prediction of Critical Transitions. *J. Stat. Phys.* 111, 839–861.

Zhang, R., de S.Cavalcante, H.L.D., Gao, Z., Gauthier, D.J., Socolar, J.E.S., Adams, M.M., and Lathrop, D.P., 2009. Boolean chaos. *Phys. Rev. E* 80, 045202.