BDsim 1.0 User Documentation

by Natércia Fernandes

April 16, 2019

Name

BDsim stands for *BioDiesel simulator*. The last version currently released is BDsim 1.0.

Contents

The package contains the following items:

`AEmodel.m`	`intermittence.m`	`rx_rates.m`
`BDsim.m`	`isOctave.m`	`side_reactions.m`
`clog_cleaning.m`	`measurements.m`	`split.m`
`clogging_kit.m`	`Mmx.m`	`stiction.m`
`clogging.m`	`ODEmodel.m`	`system_parameters.m`
`cpmx.m`	`ODEmodel_matlab.m`	`user_settings.m`
`fouling.m`	`pid_controller.m`	`valves.m`
`inputs.m`	`Qoil.m`	`Vmolar.m`

Requirements

The user needs to have installed Matlab (version '9.4.0.813654 (R2018a)' or later) or GNU Octave (version 4.0.0 or later). Older versions may also work, but they have not been tested. The programming platform Matlab requires a license from Mathworks while Octave is free software distributed under the terms of the GNU General Public License (GPL) and can be downloaded from `https://www.gnu.org/software/octave/`.

It is advisable (but not obligatory) to have installed the Symbolic Math Toolbox (for Matlab) / the Symbolic Package (for Octave) if the user wants to make use of the `heaviside` function in the definition of the inputs.

Short description

BDsim is a software for simulation of the continuous production of biodiesel. It is especially useful to generate physically meaningful data for further statistical analysis in research and/or pedagogical contexts. It can also be used to test control strategies and to perform dynamic simulation and optimization studies of systems/processes.

Physical system: The process simulated builds up on the one presented by Brásio et al. (2018), properly augmented with additional process units and equipment. For a more detailed description, please refer to **?**. The extended process comprises various sequential units: the filter to clean up the input oil; the transesterification reactor where the raw materials (methanol and oil) get transformed into ester; the heat exchanger to cool down the reacted mixture; the decanter where the separation into light and heavy phases takes place; the washer where the light phase is treated with water to eliminate its residues of glycerol and non-reacted methanol; and the dryer to remove the traces of water from the washing step. It is noteworthy that the model behind simulator **BDsim** differs significantly from the model developed by Brásio et al. (2018): (i) **BDsim** model includes the filtration process; (ii) **BDsim** model does not assume constant height of the interface between heavy and light phases in the decanter. Instead, it considers that height as a controlled variable (in accordance to what happens in industrial practice); (iii) The model of **BDsim** incorporates an energy balance of the heat exchanger, in order to make the simulator more realistic; (iv) The simulator features the typical sensor configuration and the system may work in open loop or in closed loop under a set of PID controllers; (v) The model of package **BDsim** accommodates the possibility of process changes/degradation and a panoply of sensor and valve faults. Among them is stiction, modelled according to the proposal of Kano et al. (2004). **BDsim** can simulate a wide variety of sensor faults and additionally it enables the possibility of intermittence in all them.

Application: The application is coded as a main script (`BDsim.m`) that calls several functions to accomplish diverse tasks: `ODEmodel.m` contains the set of Ordinary Differential Equations (ODEs) to be integrated over time describing the behaviour of the first four units of the system and the valve's dynamics; `AEmodel.m` contains the Algebraic Equations (AEs) to model the last two units; `Qoil.m` determines the instantaneous flow-rate of the oil stream fed to the reactor; `cpmx.m`, `Mmx.m`, and `Vmolar.m` compute several physical properties; `rx_rates.m` and `split.m` evaluate the chemical reaction and the separation degree of the chemical compounds in the system, respectively; `inputs.m` updates the process input variables along time via an AutoRegressive Moving Average with eXogenous inputs (ARMAX) model for external stimuli and by calling `pid_controller.m` (where Proportional Integral Derivative controllers are implemented) for the input variables manipulated by controllers; function `valves.m` updates the input signal of valves, taking into consideration their possible malfunction; `measurements.m` evaluates the measured variables from the state and the input variables of the process taking into account possible faults affecting the sensors. Functions `clogging_kit`, `clogging.m`, and `clog_cleaning` are responsible for the description of the oil filter clogging and its cleaning procedure; functions `fouling.m`, `intermittence.m`, `side_reactions.m`, and `stiction.m` implement other deterioration/faults suffered by the system, namely formation of fouling/scale in the heat exchanger, intermittences in sensor signals, existence of side reactions besides transesterification, and stiction in valves. Additionally, `system_parameters.m` works as an input file where the user can define/modify the characteristics and properties of the physical system (kinetic parameters, physical properties of chemical compounds, raw materials streams characterization, geometry of the system, and equipment parameters). The user is advised to modify this file only if s/he has a deep knowledge on biodiesel production and is able to perform consistent modifications. Otherwise, use the chemical system provided, that is, do not modify file `system_parameters.m`.

More importantly, `user_settings.m` works as the main input file for the user. It allows to specify the initial state of the system, stimulate the system along time, and set the parameters that define the controllers and the measurements devices. Moreover, it allows the user to specify the faults affecting the process, the sensors, and the valves. The chemical-physical system suffers the action of external disturbances simulated by ARMAX(1,1,1) models (characterized by a set of parameters provided by the user in `user_settings.m`). Simultaneously, the system is under the action of a number of PID controllers that can be either in manual or automatic modes. The control loops can be set by the user in the script `user_settings.m`; however, the user is advised to modify them only in the context of control studies of the system; i.e., for data generation purposes, the user is advised to maintain the control strategy as originally designed.

BDsim simulator accommodates, for each control loop, the possibility of operation in open or closed loop by simply setting appropriate parameters in `user_settings.m`. The application computes the evolution of the system from its initial state in response to the suffered disturbances/loads and the control actions taken along time. A summary of the results is shown graphically. The complete simulation results are saved in the folder `results` in four ASCII files for the user's convenience and disposal: `inputs`; `measurements`; `setpoints`; and `states`. The names are suggestive of their contents.

Usage

Downloading the package: **BDsim** is released under the GPL license and can be downloaded from `https://github.com/naterciafernandes/BDSIM`.

Setting the characteristics of the physical system: The characteristics of the chemical-physical system are set in the file `system_parameters.m` and include the kinetic parameters of the transesterification reaction, several physical properties (such as density, molar mass, and specific heat capacity) of the chemical compounds present in the system, composition and physical properties of the input streams (oil stream and methanol+catalyst stream), geometry of the system (volume of the reactor, cross-sectional area and height of the decanter, and thickness of the membrane used in the oil filter unit), and equipment characteristics (valves specifics: gains, time constants, maximum flow-rate allowed by the valve placed downstream of the decanter, and valve constant of the oil valve; pump head; filter cartridge specifics: size and number of its pores).

The user of **BDsim** can edit the file `system_parameters.m` with an editor of his/her preference to modify the chemical-physical system. The units of the parameters are indicated in the file for his/her reference. However, the user is *advised to carry out changes of this file only if s/he has a reasonable knowledge of the process* and is able to perform consistent modifications. The user could, for example, change the type of oil (which would call for updated kinetic parameters; density, molar mass, and specific heat capacity values for TG, DG, MG, and E components; and oil viscosity) or modify the composition of the input streams (which also requires a coherent change of their physical properties)). But in most part of the cases, the user does not have to / does not need to / should not modify the file `system_parameters.m`.

Setting the simulation scenario: The user is expected to specify the scenario under which the simulation should be carried on. In particular, s/he should establish the initial state of the system and provide information to define the stimuli to the system along the simulated experiment. The user should also switch on or off the controllers and set regulator or servo control. Additionally, the user should characterize all process, sensors, and actuators faults/degradation phenomena. In order to set up the simulation conditions, the user's input to **BDsim** simulator is performed by editing the file `user_settings.m` according to the guidelines below.

Guidelines to set the intended scenario in file `user_settings.m`:
To simplify the task, take the example file provided in the package **BDsim** as a reference and modify as needed. For the user's convenience, the file is organized in blocks.

Block 1: Settings related to the simulation time.

```
1
2 % Time ----------------------------------------------------------------------
3 %---------------------------------------------------------------------------
4     ti = 0;                                    % initial time, s
5     tf = 260000;                               % final time, s
6     dt = 5;                                    % time interval, s
7
8     t = (ti:dt:tf)';                           % time vector, s
9     lt = length(t);                            % length of the time vector
```

Indicate here the initial time (usually zero), final time of the simulated experiment, and sampling time (all in seconds). Do not modify lines 8 and 9.

Block 2: Initial state.

```
11
```

```
% Initial state ----------------------------------------------------------------
%-------------------------------------------------------------------------------
    % NOMENCLATURE:
    %    x - molar fraction (-)
    %    T - temperature (K)
    %    h - level (m)
    %    r - radius of pores (m)
    %    lift - valve lift (stem position) (m)
    %
    %    F - filter
    %    R - reactor
    %    D - decanter
    %    L - light phase (decanter)
    %    H - heavy phase (decanter)
    %    TG - triglycerides
    %    DG - diglycerides
    %    MG - monoglycerides
    %    M  - methanol
    %    E  - ester
    %    G  - glycerol
    %
    % EXAMPLE: xRTG is the molar fraction of TG in the reactor.

                % Reactor
                %  1-xRTG    2-xRDG     3-xRMG    4-xRM    5-xRE     6-xRG          7-TR
    sv0( 1: 7) = [0.002455  0.000553  0.0000467  0.42353  0.43022   0.14319   60.4+273.15];

                % Decanter (light phase)
                %    8-xLTG      9-xLDG      10-xLMG      11-xLM      12-xLE      13-xLG
    sv0( 8:13) = [4.2864e-03 9.6570e-04 8.1554e-05 2.4287e-01 7.5098e-01 8.1550e-04];

                % Decanter (heavy phase, interface level, and temperature)
                %    14-xHM      15-xHE      16-xHG   17-hH       18-TD
    sv0(14:18) = [6.6574e-01 1.5852e-04 3.3410e-01    0.5   50+273.15];

                % Filter
                % 19-r
    sv0(19)    = [pp.rclean];

                % Valves
                % 20-lifto  21-liftH
    sv0(20:21) = [      40      28.5];

    lsv = length (sv0);                                      % # of state vrs
```

Variable `sv0` represents the initial state of the system and comprises the following 21 state variables:

1. molar fraction of triglycerides in the reactor [–], `xRTG`;
2. molar fraction of diglycerides in the reactor [–], `xRDG`;
3. molar fraction of monoglycerides in the reactor [–], `xRMG`;
4. molar fraction of methanol in the reactor [–], `xRM`;
5. molar fraction of ester in the reactor [–], `xRE`;
6. molar fraction of glycerol in the reactor [–], `xRG`;
7. temperature in the reactor [K], `TR`;
8. molar fraction of triglycerides in the light phase of the decanter [–], `xLTG`;

9. molar fraction of diglycerides in the light phase of the decanter [–], `xLDG`;
10. molar fraction of monoglycerides in the light phase of the decanter [–], `xLMG`;
11. molar fraction of methanol in the light phase of the decanter [–], `xLM`;
12. molar fraction of ester in the light phase of the decanter [–], `xLE`;
13. molar fraction of glycerol in the light phase of the decanter [–], `xLG`;
14. molar fraction of methanol in the heavy phase of the decanter [–], `xHM`;
15. molar fraction of ester in the heavy phase of the decanter [–], `xHE`;
16. molar fraction of glycerol in the heavy phase of the decanter [–], `xHG`;
17. height of the interface between the heavy and the light phases in the decanter [m], `hH`;
18. temperature in the decanter [K], `TD`.
19. radius of the filter pores [m], `r`.
20. lift of the heavy phase valve [%], `liftH`.
21. lift of the oil valve [%], `lifto`.

Provide the initial state `sv0`, which may be a steady or a non-steady state point. The initial state indicated in the example file is an approximately steady-state point (for a completely clean filter), but this is not obligatory. Do not modify line 55 in the snippet above.

Block 3: Initial input variables.

```

% Initial input variables ------------------------------------------------------
%------------------------------------------------------------------------------
   %          1         2     3         4      5      6
   %      vinputo      Tmet  Fmet      Toil  Qheat  vinputH
   %          %         K   kg/h        K      W      %
   u0 =     [40.0  50+273.15  657  60+273.15  23615    28.5];   % initial input values

   lu = length (u0);                                           % # of input vrs
```

Variable `u0` contains the initial values for all the six input variables (including disturbances/load variables and manipulated variables), referred to as follows:

1. input signal of the oil valve [%], `vinputo`;
2. temperature of methanol input stream [K], `Tmet`;
3. flow-rate of methanol input stream [kg/h], `Fmet`;
4. temperature of oil input stream [K], `Toil`;
5. heat withdrawn at the heat exchanger [W], `Qheat`;

6. input signal of the heavy phase valve [%], vinputH;

The values provided in the example file are the corresponding to the steady-state provided in sv0, but this is not obligatory. You can supply a different set of initial input values u0. Do not modify line 65 of the snippet above.

Block 4: Settings related to external stimuli: ARMAX parameters and exogenous inputs.

```
67
68 % Disturbances (ARMAX settings for: loads + manipulated vrs of loops in MANUAL) ----------
69 %--------------------------------------------------------------------------------------------
70     % NOTE:
71     %    For inputs that are manipulated vrs under a control loop in AUTO mode,
72     %    the ARMAX model doesn't apply: phi, theta, eta, unoise_std, and d set
73     %    by the user won't be taken into consideration.
74
75   % ARMAX (1, 1, 1) parameters
76     %                    1      2      3     4      5      6
77     %               vinputo  Tmet   Fmet  Toil  Qheat  vinputH
78     armax.phi        = [   0    .05    .03     0    .01      0];  % weight for input at t-1
79     armax.theta      = [   0    .06    .07     0    .00      0];  % weight for noise at t-1
80     armax.eta        = [   0    .95    .97     0    .98      0];  % weight for exogenous input
81     armax.unoise_std = [   0    .10    .01     0    .01      0];  % noise standard deviation
82
83   % Exogenous inputs for disturbances along time
84     d = u0 .* ones(lt, lu);                                     % defaults (constant) from u0
85     d(:,2) = d(:,2) +3*sin(pi/(12*3600)*t);                     % daily changes for Tmet, K
86     d(:,5) = d(:,5) +1000*heaviside(t-100000);                  % deliberate step in Qheat, W
```

In this block, the user establishes the stimulation of the system by providing the ARMAX models parameters and the exogenous inputs. These models will be used to compute the load variables and the "manipulated" variables whose controllers are turned to manual mode (see Block 5). Characterize the ARMAX(1,1,1) models used to compute the external stimuli to the system by setting the elements of structure armax for each input variable: the coefficient associated to the input variable of the previous time instant, phi; the coefficient of the noise of the previous time instant, theta; the coefficient of the noise of the previous time instant, eta; and the standard deviation of the noise affecting the variable, unoise_std. All of them are vectors with as many elements as the input variable. Complement this information with the exogenous input profiles, d, as follows. Matrix d contains as many profiles along time as input variables (each column of d is a profile). Line 84 of the snippet sets its default values (as constant profiles along time) from the initial value of the input variables vector, u0, entered by the user as explained above. Do not modify this line. To perform changes, it is recommended to maintain the defaults line intact and only later ahead overwrite one or more of the default profiles provided. This procedure is exemplified in lines 85-86 of the snippet, where normal operation conditions are mimicked for the temperature of the inlet methanol stream (daily periodic changes) and a deliberate step change of 1000 W is imposed by the plant operator at t = 100 000 s.

Note 1: if armax.eta is zero (for a certain input variable), the corresponding exogenous input won't have any effect. Notice that if the user wishes to substitute the ARMAX(1,1,1) model by an ARMA(1,1,1) model, it is enough to set the parameter armax.eta to zeros (eliminating the contribution of the exogenous inputs).

Note 2: even if the ARMAX parameters and the exogenous input are set to non null values, for a given input variable, the ARMAX model will only apply if that input variable is not under the responsibility of a controller in automatic mode (see Block 5). I.e., if an input variable belongs to a closed control loop (`mode=1`), then the ARMAX parameters and `d` (even if set by the user to non nulls) won't apply to that input variable. They apply only to disturbances/load variables and to "manipulated" variables of loops in manual mode.

Block 5: Control settings.

```
 88
 89 % Control settings ------------------------------------------------------------
 90 %-------------------------------------------------------------------------------
 91
 92   % Loop mode, setpoint, measured vr, manipulated vr, & controller parameters
 93   % (as many as control loops)
 94     sp1 = (60.4+273.15)*ones(lt, 1);             % sp profile for controlled vr TR, K
 95     sp2 = (50+273.15)*ones(lt, 1);               % sp profile for controlled vr TD, K
 96     sp3 = 0.5*ones(lt, 1);                       % sp profile for controlled vr hH, m
 97     sp4 = 3000*ones(lt, 1) +100*heaviside(t-5000);% sp profile controlled vr Foil, kg/h
 98
 99         %  control loop:        1        2       3       4
100         %  controlled vr:      TR       TD      hH    Foil
101         %                       K        K       m    kg/h
102         %  manipulated (u): Toil    Qheat vinputH vinputo
103         %                       K        W       %       %
104     mode            = [         1        0       1       1]; % loop mode (MANUAL/AUTO)
105     sp              = [       sp1      sp2     sp3     sp4]; % loop setpoint profile
106     pvindex         = [         1        2       3       4]; % loop measurement (pv index)
107     uindex          = [         4        5       6       1]; % loop manipulated (u index)
108     pid.kc          = [        12   -10542   -1315  3.4e-3]; % [u]/[pv]
109     pid.taui        = [     16960     8350    3600      15]; % s
110     pid.taud        = [         0        0       0       0]; % s
111     pid.lower_bound = [30+273.15        0       0       0]; % [u]
112     pid.upper_bound = [65+273.15    40000     100     100]; % [u]
113
114   % Number of time intervals to get a new control action
115     nic = 4;
```

This block contains the control related parameters. According to the control strategy presently designed for the system, four control loops are established:

1. controls `TR` by manipulating `Toil`;
2. controls `TD` by manipulating `Qheat`;
3. controls `hH` by manipulating `vinputH`;
4. controls `Foil` by manipulating `vinputo`;

Each of the control loops can be switched either to automatic (AUTO) or to manual (MANUAL) mode by setting `mode` to 1 or to 0, respectively (line 104 of the snippet). In the snippet, loops 1, 3, and 4 are switched to automatic mode while loop 2 is switched to manual mode. When a loop is in manual mode, the controller keeps mute not issuing any order to the actuator. In this case (open-loop situation), the corresponding input variable is assigned instead by an ARMAX model for disturbances (see Block 4). Variable `sp` in line 105 of the snippet is a matrix containing the set-point profiles along time for all the controllers, independently of their working mode. Each column of `d` represents the set-point profile of a single controller (previously set in lines 94–97 of the snippet). A constant column (i.e., a constant set-point) produces a regulatory control problem. If the user wants to activate a servo control situation, a proper variable profile of `sp` (for example, the Heaviside function) should be specified here. The snippet exemplifies servo control for loop number 4 (loop to control `Foil`, with a sudden increase in the set-point at `t` = 5000 s) and regulatory control for the first three loops (loops to control `TR`, `TD`, and `hH` and that maintain their set-points). Notice that the set-point profile for a certain loop won't apply if the control loop is open (manual mode) since the controller is disconnected. However, a vector of the same length as the time vector should be defined in order to build correctly matrix `d` (see line 105 of the snippet above). Variable `pvindex` (line 106) indicates which are the measured variables used in each control loop. It contains exclusively integers between 1 and `nsensors` (see Block 6). Not all the measurements (see their definition in Block 6) are necessarily included in a control loop (with the present control design, `pv(5)` is not used in any control loop; for more details about the purpose of this measurement see explanations about Block 7). Variable `uindex` (line 107) indicates which input variable `u` works as manipulated variable for each control loop. For example, with the present control strategy, loop 1 uses measured variable number 1, `pv(1)`, (since `pvindex(1)=1`) and manipulates input variable number 4, `u(4)`, (since `uindex(1)=4`). That is, the control of `TR` in loop 1 is achieved by measuring `TR` and manipulating `Toil`. The structure `pid` (lines 108–112) contains the parameters of all the PID controllers (including bounds for the manipulated variables). The structure field names make their content obvious. All the fields of the `pid` structure are vectors with a number of elements given by the number of control loops (both in manual and in automatic modes). The derivative action for all the controllers was set to zero, i.e., the controllers are working as PI's rather than pure PID's. If the user wants to design a new control strategy for the process (just for users with control studies purposes), there is need to revise these parameters in what concerns their number and values. It is noteworthy that adding a new controller requires its tuning by the user. Set variable `nic` (line 115 of the snippet), which is an indication of the frequency under which the controllers issue orders (necessarily lower, or at maximum equal, than the "continuous" process time interval, `dt`). It is defined as the number of time intervals between consecutive releases of control orders. If the process time interval `dt` = 30 s and `nic` = 4, it means that the controllers are issuing new orders every 30 s×4 = 120 s.

Note: for using **BDsim** as a benchmark simulator for data generation, the user is advised to not change the control policy (i.e., do not modify lines 106–112 of the snippet). Most part of the times, this user will also want to keep the all 4 loops in automatic mode (`mode=1`).

Block 6: Settings for activation/deactivation of sensors faults.

```
117
118 % Sensors faults activation/deactivation ---------------------------------------
119 %--------------------------------------------------------------------------------
120     % NOTE: Possible faults affecting the sensors and how to set them:
121     %                pv = signal * [ a*v +b +noise_std*randn(1) ]
122     %       where pv - a sensor output (measured variable)
123     %             v - a sensor input (function of sv and u)
124     %
125     % For signal = 1 (sensor is "alive"):
126     %       scaling:       a = constant!=1 or a(t) &  b = 0          with 0<a<infinity
127     %       drift:         a = 1                 &  b = b(t)
128     %       bias:          a = 1                 &  b = constant     with b<0 or b>0
129     %       stuck sensor:  a = 0                 &  b = constant     min < b < max
```

```
  %       pure noise:     a = 0                 &  b = 0
  %
  % For signal = 0 (sensor is "dead"):
  %       loss of signal: a = any               &  b =  any       (a & b don't apply)


  nsensors = 5;                             % # of sensors
                                            % nsensors has to be >= # of control loops

% Default faults (none)
  sig = ones (lt , nsensors);
  a = ones (lt , nsensors);
  b = zeros (lt , nsensors);
  sfaults.isIntermit = zeros (1, nsensors);

% Setting up faults (by overwriting the defaults above)
  b(:,2) = 5e-6*(t -100000).*heaviside (t  -100000);              % drift in TD sensor

  b(:,5) = 5e3*( heaviside (t- 60000)     ...                    % a bias in
               -heaviside (t -80000) );                          % sensor 5 (DPfilter)


               % sensor       1        2        3        4       5
    % measured vr (pv):      TR       TD       hH     Foil  DPfilter
    %                         K        K        m     kg/h      Pa
  sfaults.signal     = [sig(:,1) sig(:,2) sig(:,3) sig(:,4) sig(:,5)];     % -
  sfaults.a          = [  a(:,1)   a(:,2)   a(:,3)   a(:,4)   a(:,5)];     % [pv]/[sv]
  sfaults.b          = [  b(:,1)   b(:,2)   b(:,3)   b(:,4)   b(:,5)];     % [pv]
  sfaults.isIntermit = [       0        1        0        0        0];     % -
  sfaults.tmaxInterm = [       0     3600        0        0        0];     % s
  sfaults.noise_std  = [     0.1      0.1     5e-3        5      300];     % [pv]
```

This is the block where the user sets the measurement characteristics and can trigger faults in the system sensors, therefore introducing sources of variability. The measured variables, pv, are the process variables

1. temperature in the reactor [K], TR;
2. temperature in the decanter [K], TD.
3. height of the interface between the heavy and the light phases in the decanter [m], hH;
4. mass flow-rate of the oil inlet stream [kg/h], Foil;
5. pressure drop across the filter [Pa], DPfilter,

deteriorated to some degree with noise and, possibly, with other faults such as scaling, drift, bias, hard faults (stuck sensor or lack of signal) and intermittences. All of these faults can last for the whole simulated experiment or can be framed in time. For a certain time instant, variable pv measured by a certain sensor is defined as `pv = signal * [a*v +b +noise_std*randn(1)]`, where v is the sensor input variable (which is a function of the state variables, sv, and of the process input variables, u, computed according to measurements.m); signal indicates whether an electric signal from the sensor is arriving at the controller (signal = 1, sensor is alive) or not (signal = 0, sensor is dead); a is a scaling factor of the sensor's input v (depending on the fault, a might be constant or variable in time ranging from 0 to infinity); b is an additive term responsible for a positive or negative translation of the reading (a bias if `b` is constant or a drift if `b` is a monotonous function of time); and `noise_std` is the standard deviation of the noise that corrupts the sensor signal. The set of faults referred to above can be activated/deactivated with a proper choice of these parameters summarized in Table 1. The intermittence can be any of the other faults, but appearing/disappearing at a random frequency.

Table 1: Sensor faults setting parameters.

Fault	`signal`	`a`	`b`	**Notes**
scaling	1	`a` $\neq 1$	0	`a`=constant or `a=a(t)`, $0 <$ `a` $< \infty$
drift	1	1	`b(t)`	monotonous, positive or negative
bias	1	1	constant	`b` $\neq 0$, positive or negative
pure noise	1	0	0	intensity: standard deviation
stuck sensor	1	0	constant	minimum $<$ `b` $<$ maximum
loss of signal	0	any	any	`a` and `b` do not apply
intermittences	(any of the faults above)			signature: appearance/disappearance at a random frequency

The number of sensors installed in the system (five) is specified in line 136 of the snippet. The user must not modify this value (`nsensors` = 5) unless s/he wants to install extra sensors in the system. Such a task, however, would require further changes in the code (namely in function `measurements.m`) to append the information about the extra measurement(s) furnished by the added sensor(s). It is noteworthy that, for the designed control strategy, not all the sensors are incorporated in control loops: measured variable `pv(5)` (i.e, `DPfilter`) does not belong to any control loop. Lines 139–143 of the snippet define the default values of sensor faults. The defaults correspond to a situation without faults, but with the usual noise of the sensor readings. Do not modify the defaults (lines 139–143 of the snippet). The user might introduce faults (if any) right after the definition of the defaults, overwriting them. For that, take into consideration the information given in Table 1. The example file `user_settings.m` provided in the package (see lines 145–149 of the snippet) illustrates two faults:

- a positive linear drift (at a rate of 5×10^{-6} K/s) for sensor number 2 (`TD` sensor) since `t` = 100 000 s on;
- a positive bias of 5×10^{3} Pa for sensor number 5 (`DPfilter` sensor) between 60 000 s and 80 000 s.

But the number of faults, their characteristics, and which are the affected sensors are to be decided by the user. With the exception of the of noise standard deviation (which is constant along the simulation), each sensor parameter is a vector along time rather than a single value. Therefore, the user can easily limit the duration of a fault or even program a sequence of faults for the same sensor by using the Heaviside function. Some examples follow:

- Constant scaling of 115% for sensor 2 between `t` = 10 000 s and `t` = 15 000 s:
  `a(:,2) = 1 +0.15*( heaviside (t-10000) -heaviside (t-15000) )`
  keep the defaults for `b(:,2)` and `signal(:,2)`

- Constant scaling of 75% for sensor 2 between `t` = 14 000 s and `t` = 18 000 s:
  `a(:,2) = 1 -0.25*( heaviside (t-14000) -heaviside (t-18000) )`
  keep the defaults for `b(:,2)` and `signal(:,2)`

- Linear drift (down, at a rate of 5×10^{-8} meters per second) for sensor 3 for all the simulation time:
  `b(:,3) = -5e-8*(t-ti) .* ( heaviside(t-ti) -heaviside(t-tf) )`
  or, in this special case, simply `b(:,3) = -5e-8*t`
  keep the defaults for `a(:,3)` and `signal(:,3)`

- Negative bias of 2℃ (or 2 K) for sensor 1 between `t` = 1800 s and `t` = 18 000 s:
  `b(:,1) = -2*( heaviside (t-1800) -heaviside (t-18000) )`
  keep the defaults for `a(:,1)` and `signal(:,1)`

- Sensor 1 gets stuck with the value of 333K at `t` = 18 000 s and does not recover:
  `b(:,1) = 333*heaviside (t-18000)`
  `a(:,1) = 1 -heaviside (t-18000)`
  keep the defaults for `signal(:,1)`

- Sensor 2 suffers a loss of signal at `t` = 70 000 s but it recovers 1 hour later:
  `signal(:,2) = 1 -heaviside (t-70000) +heaviside (t-73600)`
  keep the defaults for `a(:,2)` and `b(:,2)`
  (you may change them while the fault lasts but they will not apply).

Rather than changing directly lines 155–157 of the snippet to overwrite the defaults (when applicable), the user is advised to prepare individually any wished sensor fault(s) before the establishment of structure `sfaults` (lines 152–160 of the snippet example). Then, in lines 155–157 of the snippet, such individual elements will be integrated at once in the required vectors `signal`, `a`, and `b` of structure `sfaults`. To set an intermittent fault for a certain sensor, the user must set a fault (at choice) as if it was a “continuous” fault and then flag its intermittence (by setting `sfaults.Intermit=1`, line 158 of the snippet). Additionally, the user should set the maximum duration of a single instance of that intermittent fault, `sfaults.tmaxInterm` (line 159 of the snippet), which is a parameter that relates to the frequency of the intermittence. On the other hand, if `sfaults.Intermit=0` (no intermittence), the fault set by the user will be treated as “continuous”, that is, it will apply for all the time instants for which it is active (which might coincide or not with the overall simulation window) according to the profiles set for parameters `a`, `b`, and `signal` for that sensor. For example:

- Sensor 2 suffers an intermittent loss of signal at `t` = 70 000 s but recovers 1 hour later; the longest instance of intermittence is always smaller or equal than 500 s:
  `signal(:,2) = 1 -heaviside (t-70000) +heaviside (t-73600)`
  keep the defaults for `a(:,2)` and `b(:,2)`
  (you may change them while the fault lasts but they will not apply)

flag intermittence on for sensor 2: `sfaults.isIntermit = [0 1 0 0 0]`
set the maximum duration of a single instance of intermittence:
`sfaults.tmaxInterm = [0 500 0 0 0]`
(in this example, sensors 1, 3, 4, and 5 were taken as non-intermittent).

In the example of the snippet, sensors 1, 3, and 4 don't experience any fault along the whole simulation experiment and sensor 5 experiences a permanent fault between 60 000 s and 80 000 s while sensor 5 experiences an intermittent fault since 100 000 s on, for which each of its instances lasts a maximum of 3600 s. The Block finishes with the setting of the noise standard deviation associated to each sensor reading (line 160 of the snippet).

Block 7: Settings for activation/deactivation process deterioration/faults.

```
162
163 % Process deteriorations/faults activation/deactivation -----------------------------
164 %-------------------------------------------------------------------------------------
165   % Default faults (none)
166     pfaults.clog_fraction = 0;
167     pfaults.DPclean = Inf;
168     pfaults.ratio_robs_r = 1;
169     pfaults.fouling = 0;
170     pfaults.foulingpar = [1];
171
172   % Clogging of the oil filters
173     pfaults.DPclean = 1e5;                 % DPfilter at which cleanings are done, Pa
174     pfaults.clog_fraction = 5.95e-7;       % fraction adsorbed to pores walls, -
175     pfaults.filter_std = 5e-15;            % standard deviation for zF/nF, m
176
177   % Catalyst activity loss by consumption in side reactions
178     pfaults.ratio_robs_r = .9;             % robserved/r, dimensionless
179
180   % Fouling in the heat exchanger
181     pfaults.fouling = 1;                   % 0: no fouling; 1: linear fouling
182     pfaults.foulingpar = [3e-7];           % changing rate of the fouling resistance
```

In this block, the user can add process deterioration / faults. It is possible to activate/deactivate: the clogging of the oil filters (which affects directly the oil feed flowrate); the catalyst activity loss caused by its consumption in side reactions such as saponification (which slows down the transesterification reaction); and the fouling/scale formation in the heat exchanger (which difficults the cooling of the mixture leaving the reactor). The block begins with the setting of the process deterioration defaults (lines 165–170 of the snippet), which correspond to the situation without process degradation / faults. Do not modify these lines. To deactivate a process fault, it is enough to maintain the default for that specific phenomenon (delete or comment out any further line overwriting it). In opposition, to activate one or more process faults there is need to overwrite their defaults (as it happens in lines 172–182 of the example in the snippet). Filters clogging activation (lines 172–175 of the snippet): (i) specify the pressure drop across the filter at which the filter should be cleaned/substituted. Due to the pores clogging the pressure drop across the filter increases monotonously until the filter is cleaned. To ensure the performance and safety of the industrial process, such procedure occurs usually at 1×10^5 Pa. But the user can set other values to simulate a too early or a too late cleaning (by indicating a `DPclean` smaller and bigger than 1×10^5 Pa, respectively); (ii) indicate the intensity of the clogging phenomenon, `clog_fraction`. The value 5.95×10^{-7} was chosen according to the typical behaviour of the system. The user might experiment with different values (decreasing `clog_fraction` reduces the intensity of the clogging; in the limit, when it is set to zero (i.e., the default) there is no clogging at all). This parameter depends on the type and conditions of the processed oil; (iii) specify the variability from filter to filter introduced in the cleaning procedure by setting parameter `filter_std`, which represents the standard deviation associated to the ratio of the filter thickness and its number of pores, zF/nF. Catalyst activity loss activation (lines 177–178 of the snippet): specify the ratio between the observed reaction rate and the ideal reaction rate, $0 <$ `pfaults.ratio_robs_r` < 1. The catalyst can get consumed in side reactions such as saponification, therefore decreasing the effective transesterification reaction rates. Fouling/scale activation (lines 180–182 of the snippet): flag `pfaults.fouling` = 1 and specify a value for the linear fouling formation rate (`pfaults.foulingpar`).

Block 8: Settings for activation/deactivation of valve faults.

```
184
185 % Valves faults activation/deactivation ------------------------------------------
186 %---------------------------------------------------------------------------------
187   % Stiction
188     % S - magnitude of the (deadband)+(slip-jump)
189     % J - magnitude of the slip-jump
190
191     % J=0 & S=0: no stiction
192     %       J=0: pure deadband
193     %       J<S: undershoot case with jump
194     %       J=S: no offset case (no deadband)
195     %       J>S: overshoot case
196
197               % valves  1  2
198     vfaults.S      = [  0  0 ];              % deadband+jump for each valve, %
199     vfaults.J      = [  0  0 ];              % jump for each valve, %
200     vfaults.uindex = [  6  1 ];              % index of "u" containing the
201                                              % controller order, for each valve
202     nvalves = length(vfaults.uindex);
```

Finally, in this block, the user can activate/deactivate stiction in valves. At the present version of **BDsim**, the system is equipped with two valves:

1. valve placed in the heavy phase stream leaving the decanter
   (commanded by the controller signal contained in the input variable `u(6)`);
2. valve placed in the oil feed stream to the reactor
   (commanded by the controller signal contained in input variable `u(1)`).

To activate/deactivate stiction as well as to indicate the severity of the phenomenon, it is enough to specify its two characterizing parameters (lines 197–198 of the snippet): the deadband plus slip-jump, `vfaults.S`, and the slip-jump, `vfaults.J`, both expressed in percentage. It is worthy to emphasize that the first parameter, `S`, includes both the deadband and the slip-jump. To deactivate stiction in a valve set both `S` and `J` to 0. To activate stiction: by choosing different sets of those two parameters, it is possible to set pure deadband, stiction undershoot, no offset, and overshoot cases. Deadband: specify a non-null value for `S` and simultaneously set `J` to zero; Stiction undershoot: specify a smaller value for `S` than for `J`; No offset: specify the same value for `S` and for `J`; Overshoot: set a bigger value for `S` than for `jump`. The values chosen should be reasonable. Additionally, the user should indicate with which input variable each valve is associated, in which input variable is registered the controller order for the valve (line 199 of the snippet). Do not modify line 201 of the snippet.

Running the application: **BDsim** runs in both Matlab and Octave environments. Initiate Matlab or Octave. At the command-line prompt, enter the command '`BDsim`'. Notice that the current directory should correspond to the folder with the **BDsim** package. The results will be saved in four ASCII files (`inputs`, `measurements`, `setpoints`, and `states`) lumped together into a folder named `results` and placed in the user's working folder (overwriting the four mentioned files if existent). Additionally, **BDsim** writes extra information to the standard output (display): the time instant(s) at which cleaning(s) of the filter was(were) performed (if any) and the CPU time spent with the simulation experiment.

Reporting bugs

Any bug found can be reported to the author, via email `natercia@eq.uc.pt`.

References

Brásio, A. S., Romanenko, A., and Fernandes, N. C. P. (2018). Simulation and advanced control of the continuous biodiesel process. In Pinto, A. A. and Zilberman, D., editors, *Modeling, Dynamics, Optimization and Bioeconomics III*, volume 224 of *Springer Proceedings in Mathematics & Statistics*, pages 127–146. Springer International Publishing. URL `https://doi.org/10.1007/978-3-319-74086-7_6`.

Kano, M., Maruta, H., Kugemoto, H., and Shimizu, K. (2004). Practical model and detection algorithm for valve stiction. *Proceedings of the IFAC Symposium on Dynamics and Control of Process Systems*. URL `http://www.nt.ntnu.no/users/skoge/prost/proceedings/dycops04/pdffiles/papers/54.pdf`.