Simulating Hybrid Systems in Matlab/Simulink, v0.5

David A. Copp and Ricardo G. Sanfelice
Hybrid Dynamics and Control Laboratory
University of Arizona

1 Introduction to a general hybrid system model

A hybrid system is a dynamical system with continuous and discrete dynamics. Several mathematical models for hybrid systems have appeared in literature. In this paper, we consider the framework for hybrid systems used in [2,3], where a hybrid system $\mathcal{H}$ on a state space $\mathbb{R}^n$ is defined by the following objects:

- A set $C \subset \mathbb{R}^n$ called the *flow set*.
- A function f: $\mathbb{R}^n \to \mathbb{R}$ called the *flow map*.
- A set $D \subset \mathbb{R}^n$ called the *jump set*.
- A function g: $\mathbb{R}^n \to \mathbb{R}$ called the *jump map*.
- A set $O \subset \mathbb{R}^n$ called the *state space*.

The flow map f defines the continuous dynamics on the flow set C, while the jump map g defines the discrete dynamics on the jump set D. These objects are referred to as the *data* of the hybrid system $\mathcal{H}$, which at times is explicitly denoted as $\mathcal{H} = (O, f, C, g, D)$.

2 A Simulink implementation

We consider the simulation in Matlab/Simulink of hybrid systems $\mathcal{H} = (O, f, C, g, D)$ written as

$$\mathcal{H}: \qquad x \in O, u \in \mathbb{R}^m \qquad \begin{cases} \dot{x} &= f(x,u) & (x,u) \in C \\ x^+ &= g(x,u) & (x,u) \in D. \end{cases} \tag{1}$$

Figure 1 shows a Simulink implementation proposed here.

Five basic blocks are used to define the dynamics of the hybrid system $\mathcal{H}$:

- The flow map is implemented in an *Embedded Matlab function block* executing the function `f.m`. Its input is a vector with components defining the state of the system x, and the input u. Its output is the value of the flow map f which is connected to the input of an integrator.
- The flow set is implemented in an *Embedded Matlab function block* executing the function `C.m`. Its input is a vector with components of the state of the *Integrator system* x^- and the input u^-, and its output is equal to 1 if the state belongs to the set C or equal to 0 otherwise. The minus notation denotes the previous value of the variables (before integration). The value x^- is obtained from the state port of the integrator.
- The jump map is implemented in an *Embedded Matlab function block* executing the function `g.m`. Its input is a vector with components of the state of the *Integrator system* x^- and the input u^-, and its output is the value of the jump map g.
- The jump set is implemented in an *Embedded Matlab function block* executing the function `D.m`. Its input is a vector with components of the state of the *Integrator* system x^- and the input u^-, and its output is equal to 1 if the state belongs to D or equal to 0 otherwise.
- The state space is implemented in an *Embedded Matlab function block* executing the function `O.m`. Its input is a vector with components of the state of the *Integrator system* x^- and the input u^-, and its output is equal to 1 if the state belongs to O or equal to 0 otherwise.

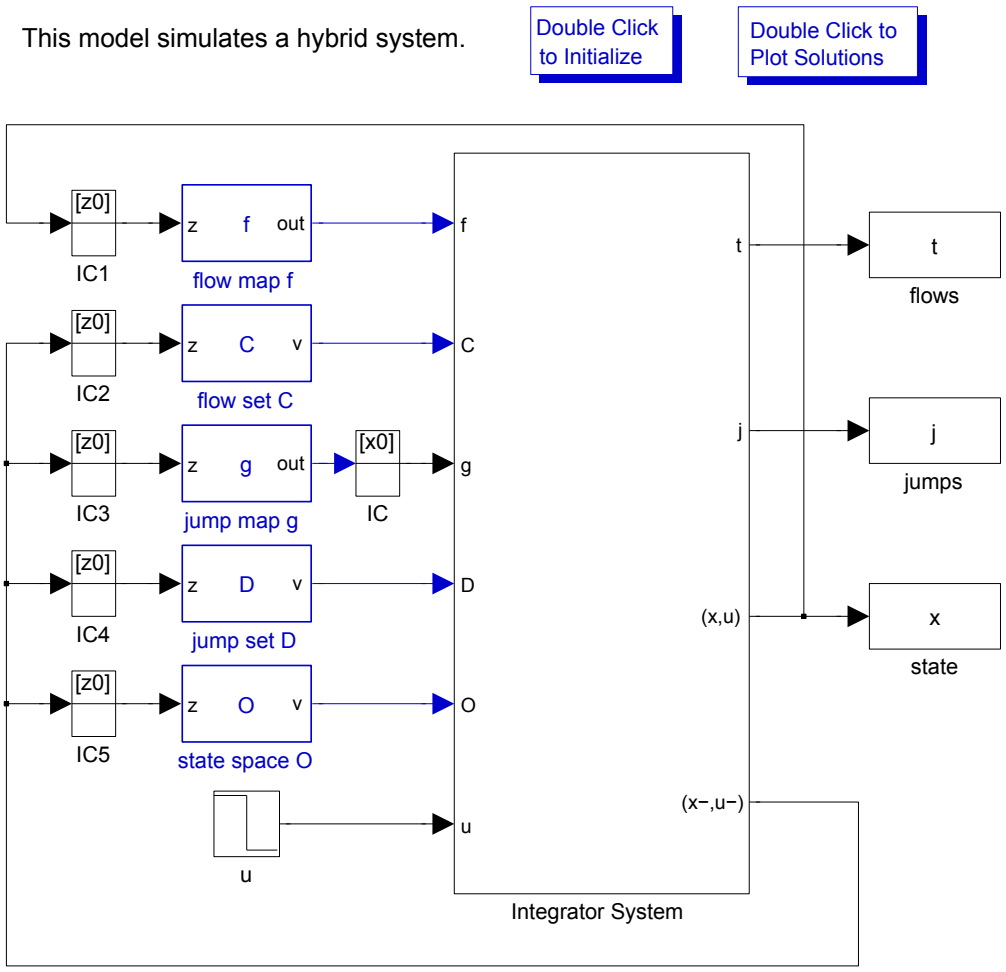

Figure 1: Matlab/Simulink implementation of a hybrid system $\mathcal{H} = (O, f, C, g, D)$ with inputs.

2.1 CT Dynamics

This block defines the continuous dynamics by assembling the time derivative of the state $[t\ j\ x^T]^T$. This is given by

$$\dot{t} = 1, \qquad \dot{j} = 0, \qquad \dot{x} = f(x, u)\ .$$

Figure 3 depicts this implementation. Note that input port 1 takes the value of $f(x, u)$ through the output of the *Embedded Matlab function block f* in Figure 1.

2.2 Jump Logic

The inputs to the jump logic block are the output of the blocks *C, D,* and *O* indicating whether the state is in those sets or not, and a random signal with uniform distribution in $[0, 1]$. Figure 4 shows that these signals, and another variable called *rule*, are the inputs of a Truth Table called *Jump Priority*.

The *Jump Priority Truth Table* includes the following logic:

```
% state
flowFlag = z(1);
jumpFlag = z(2);
stateFlag = z(3);
```

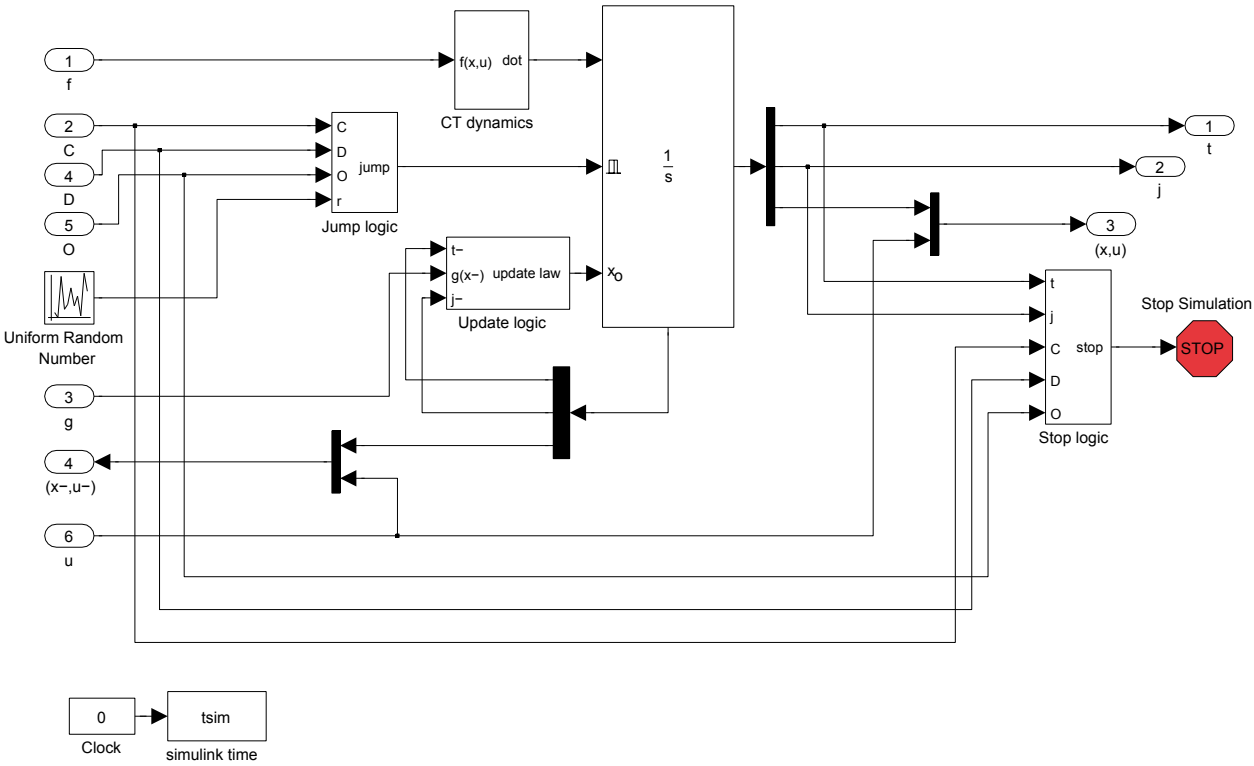

Figure 2: Integrator System

```
randomInput = z(4);

% rule = 1 -> priority for jumps
% rule = 2 -> priority for flows
% rule = 3 -> no priority, random selection when simultaneous conditions

if (rule == 1) & (jumpFlag == 1)
   out = 1;
elseif (rule == 1) & (jumpFlag == 0)
   out = 0;
elseif (rule == 2) & (flowFlag == 1)
   out = 0;
elseif (rule == 2) & (flowFlag == 0) & (jumpFlag == 0)
   out = 0;
elseif (rule == 2) & (flowFlag == 0) & (jumpFlag == 1)
   out = 1;
elseif (rule == 3)
   if (flowFlag == 1) & (jumpFlag == 0)
      out = 0;
   elseif (flowFlag == 0) & (jumpFlag == 1)
      out = 1;
   elseif (flowFlag == 1) & (jumpFlag == 1)
      if (randomInput >= 0.5)
         out = 1;
      else
         out = 0;
```

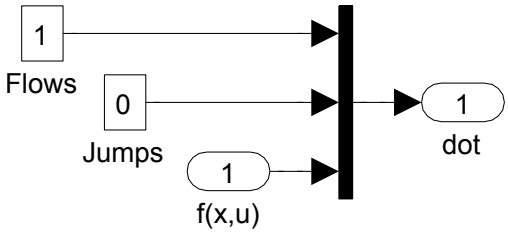

Figure 3: CT dynamics

```
      end
   else
       out = 0;
   end
end
```

The output of this Truth Table is equal to one only when the output of the *D block* is equal to one and $rule = 1$, or when the output of the *D block* is equal to one, $rule = 3$, and the random signal r is larger or equal than 0.5. Under either event, the output of this block, which is connected to the integrator external reset input, triggers a reset of the integrator, that is, a jump of $\mathcal{H}$. The reset or jump is activated since the configuration of the reset input is set to "level hold", which executes resets when this external input is equal to one (if this input remains set to one, multiple resets would be triggered).

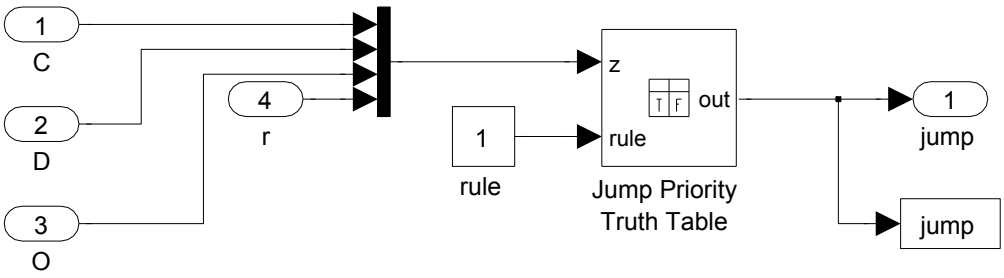

Figure 4: Jump Logic

2.3 Update Logic

The update logic uses the *state port* information of the integrator. This port reports the value of the state of the integrator, $[t\ j\ x^T]^T$, at the exact instant that the reset condition becomes true. Notice that x^- differs from x since at a jump, x^- indicates the value of the state that triggers the jump, that is, $x \in D$, while x at that same time is equal to the value assigned at the jump by the update logic. This value is given by $g(x^-, u^-)$ as Figure 5 illustrates. It also shows that the flow time t is kept constant at jumps and that j is incremented by one by the Matlab function block $j + 1$. More precisely

$$t^+ = t^-, \qquad j^+ = j^-, \qquad x^+ = g(x^-, u^-)$$

where $[t^-\ j^-\ x^{-^T}]^T$ is the state that triggers the jump.

2.4 Stop Logic

This block, shown in Figure 6, stops the simulation under any of the following events:

- The flow time is larger than or equal to the maximum flow time specified by T.

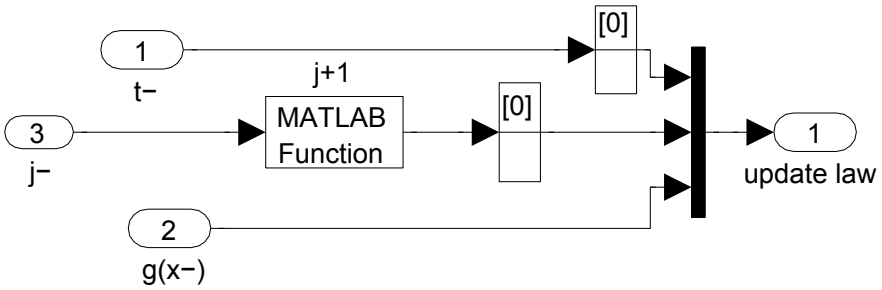

Figure 5: Update Logic

- The jump time is larger than or equal to the maximum number of jumps specified by J.
- The state of the hybrid system x is neither in C nor in D, or it is not in O.

Under any of these events, the output of the logic operator connected to the *Stop block* becomes one, stopping the simulation. Note that the inputs C, D, and O are routed from the output of the blocks computing whether the state is in C, D, and O, and use the previous value of the state x. This may cause the simulator to perform an extra iteration before stopping, but this can be resolved by plotting the solution excluding the data from the last iteration.

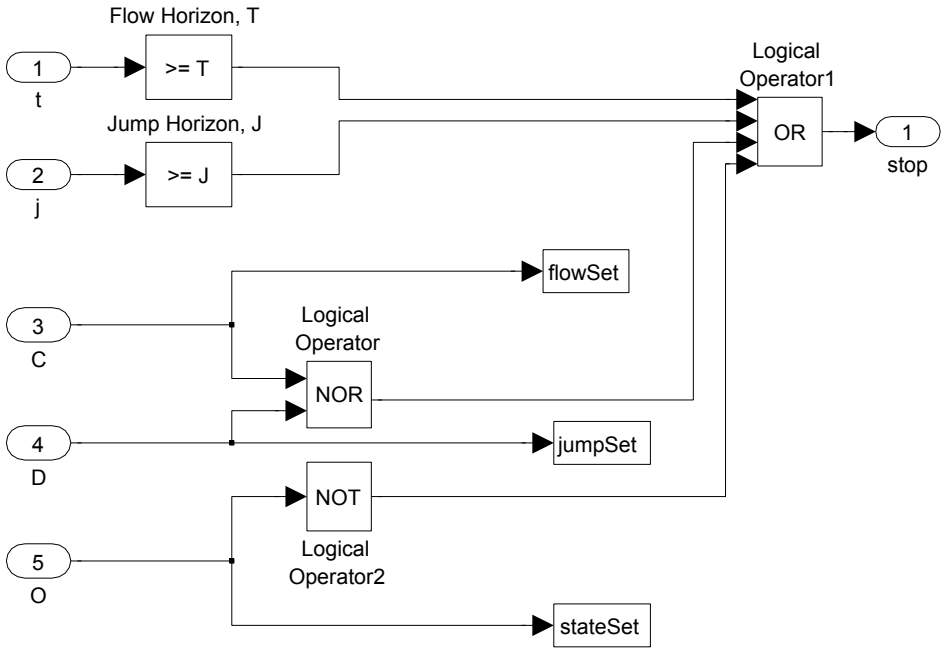

Figure 6: Stop Logic

2.5 Configuration

When the block labeled *Double Click to Initialize* is double-clicked, the simulation variables are initialized and the simulation is run by calling the script *initialization.m*. *initialization.m* defines the initial conditions by defining the initial values of the state components, any necessary parameters, the maximum flow time specified by T, the maximum number of jumps specified by J, and tolerances used when simulating. These

can be changed by editing the script file *initialization.m*. See below for sample code to initialize the bouncing ball example, Example 1.1.

```
% initialization for bouncing ball example
clear all
% initial conditions
x0 = [1;0];
u0 = 0;
% combine initial conditions
z0 = [x0; u0];
% simulation horizon
T = 10;
J = 20;
% rule for jumps
% rule = 1 -> priority for jumps
% rule = 2 -> priority for flows
% rule = 3 -> no priority, random selection when simultaneous conditions
rule = 1;
% constants
n = 2; %# of state components
m = 1; %# of input components
%solver tolerances
options = odeset('RelTol',1e-8,'MaxStep',.001);
%simulate
sim ('HybridSimulator')
```

It is important to note that variables called globally in the Embedded Matlab function blocks must be assigned values locally in the Model Workspace. This can be done in Simulink by selecting `View>Model Explorer`, opening the Model Workspace, and selecting `Add>Simulink Signal`. Use the same name as the variable called in the function block. Then the variable must be added to the data/ports in each respective Embedded Matlab function block by opening each Embedded Matlab function block and selecting `Tools>Edit Data/Ports`. Then in the Ports and Data Manager, select `Add>Data`, name the variables the same name as in the function, and set the scope to `Data Store Memory`. See the example files for this procedure. Figure 7 depicts the model workspace for the interconnection analysis example, Example 1.6.

2.6 Postprocessing and Plotting solutions

A similar prodecure is used to define the plots of solutions after the simulation is run. The solutions can be plotted by double-clicking on the block labeled *Double Click to Plot Solutions* which calls the script *postprocessing.m*. The script *postprocessing.m* may be changed to include the desired postprocessing and solution plots. See below for sample code to plot solutions to the bouncing ball example, Example 1.1.

```
%postprocessing for the bouncing ball example
% plot solution
figure(1)
clf
subplot(2,1,1),plotflows(t,j,x)
grid on
ylabel('x')
subplot(2,1,2),plotjumps(t,j,x)
grid on
ylabel('x')
% plot hybrid arc
plotHybridArc(t,j,x)
xlabel('j')
```

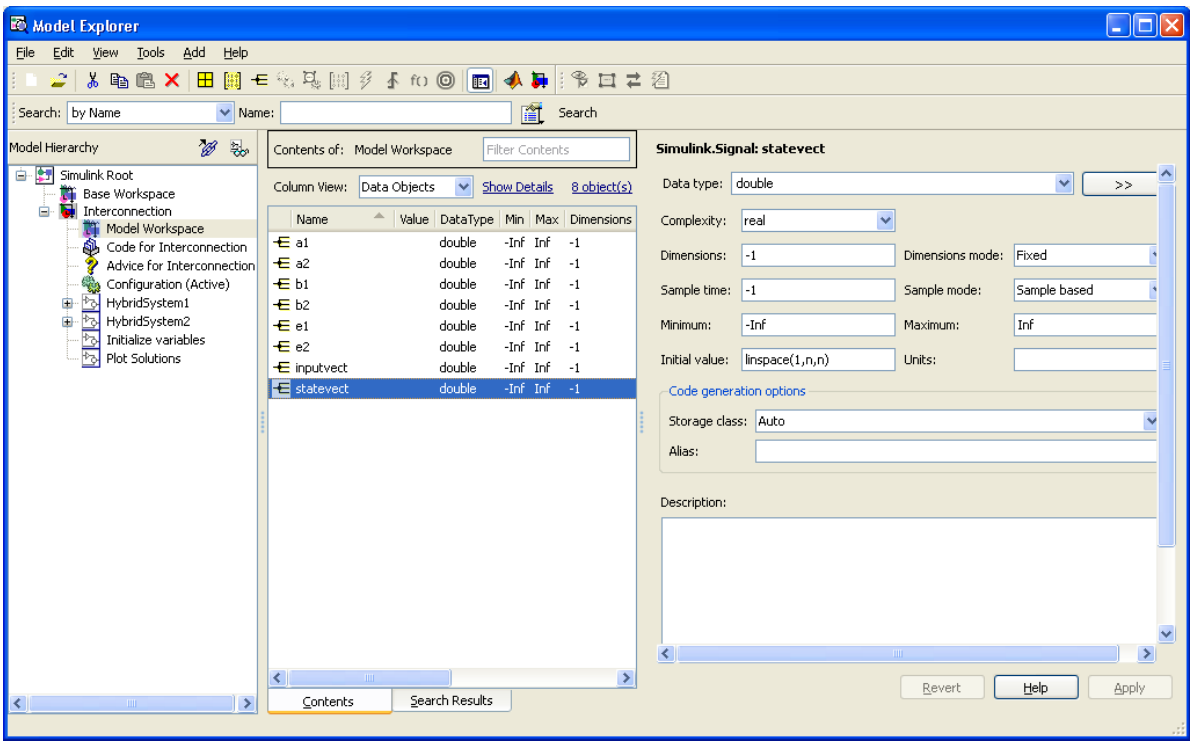

Figure 7: Model Workspace

```
ylabel('t')
zlabel('x')
```

The following functions are used to generate the plots:

- plotflows(t,j,x): plots (in blue) the projection of the trajectory x onto the flow time axis t. The value of the trajectory for intervals $[t_j, t_{j+1}]$ with empty interior is marked with $*$ (in blue). Dashed lines (in red) connect the value of the trajectory before and after the jump. Figure 8 shows a plot created with this function.
- plotjumps(t,j,x): plots (in red) the projection of the trajectory x onto the jump time j. The initial and final value of the trajectory on each interval $[t_j, t_{j+1}]$ is denoted by $*$ (in red) and the continuous evolution of the trajectory on each interval is depicted with a dashed line (in blue). Figure 8 shows a plot created with this function.
- plotHybridArc(t,j,x): plots (in black) the trajectory x on hybrid time domains. The intervals $[t_j, t_{j+1}]$ indexed by the corresponding j are depicted in the $t-j$ plane (in red). Figure 10 shows a plot created with this function.

3 Examples

The examples below illustrate the use of the implementation above.

Example 1.1 (bouncing ball with input) For the simulation of the bouncing ball system with a constant input and regular data given by

$$O := \mathbb{R}^2, f(x,u) := \begin{bmatrix} x_2 \\ -\gamma \end{bmatrix}, C := \left\{(x,u) \in \mathbb{R}^2 \times \mathbb{R} \mid x_1 \geq u\right\} \tag{2}$$

$$g(x,u) := \begin{bmatrix} u \\ -\lambda x_2 \end{bmatrix}, D := \left\{(x,u) \in \mathbb{R}^2 \times \mathbb{R} \mid x_1 \leq u \ , \ x_2 \leq 0\right\} \tag{3}$$

where $\gamma > 0$ is the gravity constant, u is the input constant, and $\lambda \in [0,1)$ is the restitution coefficient. The Matlab scripts in each of the function blocks of the implementation above are given as follows. An input was chosen to be $u(t,j) = 0.2$ for all (t,j). The constants for the bouncing ball system are $g = 9.81$ and $\lambda = 0.8$.

The following procedure is used to simulate this example:

- *HybridSimulator.mdl* is opened in Matlab/Simulink.
- The Embedded Matlab function blocks *f, C, g, D, O* are edited by double-clicking on the block and editing the script. In each embedded function block, global variables must be defined and added to the data/ports by selecting `Tools>Edit Data/Ports>Add>Data`, naming the variables the same as they are called in the function block and setting the scope to Data Store Memory. For this example, *statevect* and *inputvect* are defined in this way.
- Variables that are defined as global in the embedded function blocks must be defined locally in the Model Workspace. This is done in the Simulink model by selecting `View>Model Explorer`, dropping down the options below `HybridSimulator` and selecting `Model Workspace`. Then select `Add>Simulink Signal`. Name it the same as the variable is named in the embedded function block, and set the initial value as desired.
- The initialization script *initialization.m* is edited by opening the file and editing the script. The flow time and jump horizons, T and J are defined as well as the initial conditions for the state vector, x_0, and input vector, u_0, a rule for jumps, *rule*, and the number of state components and input components, n and m.
- The postprocessing script *postprocessing.m* is edited by opening the file and editing the script. Flows and jumps may be plotted by calling the functions *plotflows* and *plotjumps*, respectively. the hybrid arc may be plotted by calling the function *plotHybridArc*.
- The simulation stop time is set to T.
- The block labeled *Double Click to Initialize* is double-clicked to intialize the variables and run the simulation.
- The block labeled *Double Click to Plot Solutions* is double-clicked to plot the desired solutions.

```
function out = f(z)
% state
x1 = x(1);
x2 = x(2);
%input
u=u(1)
% flow map
x1dot = x2;
x2dot = -9.81;
out = [x1dot; x2dot];

function [v] = C(z)
% state
x1 = x(1);
```

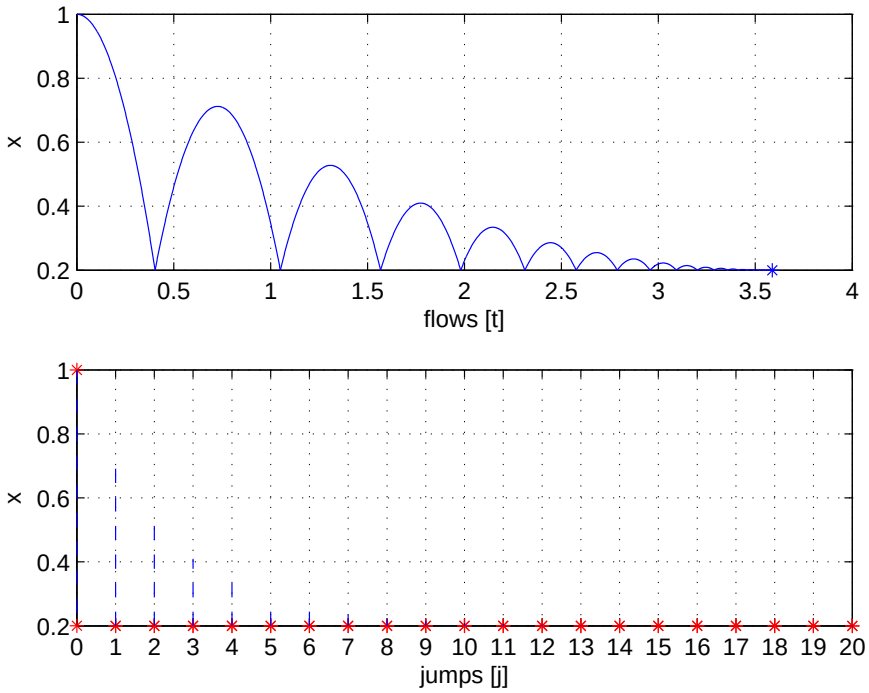

Figure 8: Solution of Example 1.1: height

```
x2 = x(2);
if (x1 >= u)  % flow condition
    v = 1;  % report flow
else
    v = 0;   % do not report flow
end

function out = g(z)
% state
x1 = x(1);
x2 = x(2);
% jump map
x1plus = u;
x2plus = -0.8*x2;
out = [x1plus; x2plus];

function [v] = D(z)
% state
x1 = x(1);
x2 = x(2);
if (x1 <= u && x2 <= 0)  % jump condition
    v = 1;  % report jump
else
    v = 0;   % do not report jump
end

function [v] = O(z)
```

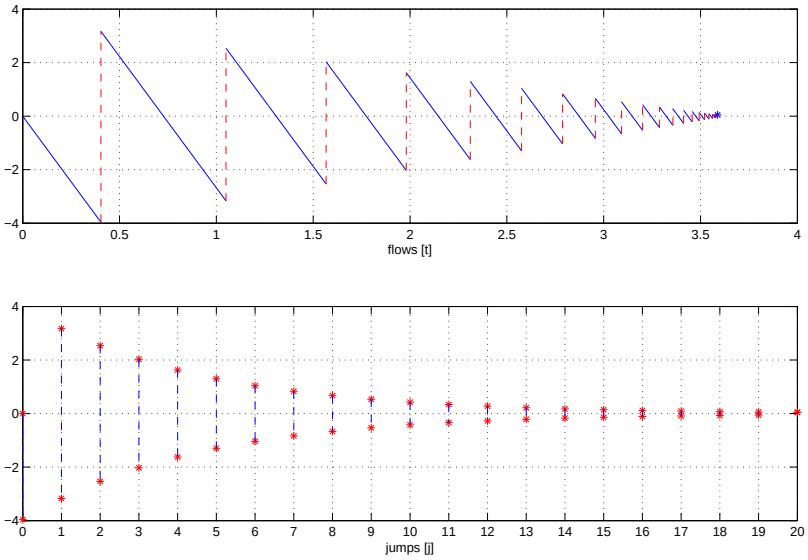

Figure 9: Solution of Example 1.1: velocity

```
v = 1;  % in the state space
```

A solution to the bouncing ball system from $x(0,0) = [1,0]$ and with $T = 10, J = 20$, $rule = 1$, is depicted in Figure 8 (height) and Figure 9 (velocity). Both the projection onto t and j are shown. Figure 10 depicts the corresponding hybrid arc.

These simulations reflect the expected behavior of the bouncing ball model. Note the only difference between this example and the example of a bouncing ball without a constant input is that, in this example, the ball bounces on a platform at a height of the chosen input value 0.2 rather than the ground at a value of 0.

For Matlab/Simulink files of this example, see Examples/Example_1.1.

□

Example 1.2 (alternative ways to simulate the bouncing ball)
a) For the simulation of the bouncing ball system with a constant input and regular data as given in Example 1.1. This example shows that a Matlab function block, such as the jump set D, can be replaced with operational blocks in Simulink. Figure 11 shows this implementation. The other functions and solutions are the same as in Example 1.1.

For Matlab/Simulink files of this example, see Examples/Example_1.2a.
b) Another way to simulate a bouncing ball (and hybrid systems in general), is to replace the integrator with resets implemented in Simulink by ODE function call with events (see, e.g., http://control.ee.ethz.ch/~ifaatic/ex/example1.m). Such an implementation gives faster simulation of a single hybrid system. A code example is presented below for the bouncing ball example. The results of this simulation are the same as those in Example 1.1.

```
% Code developed by Torstein Ingebrigtsen Bo
function [t y j] = hybridsolver( f,g,C,D,y0,TSPAN,JSPAN,rule,options,maxStepCoefficient)
% HYBRIDSOLVER solves hybrid equations
%   [t y j] = hybridsolver( f,g,C,D,y0,TSPAN,JSPAN) will integrate
%   y'=f(y) and jump by the rule y = g(y). y is a vector with the same
%   length as y0. Both must return a vector with the
%   equal length as y0.
```

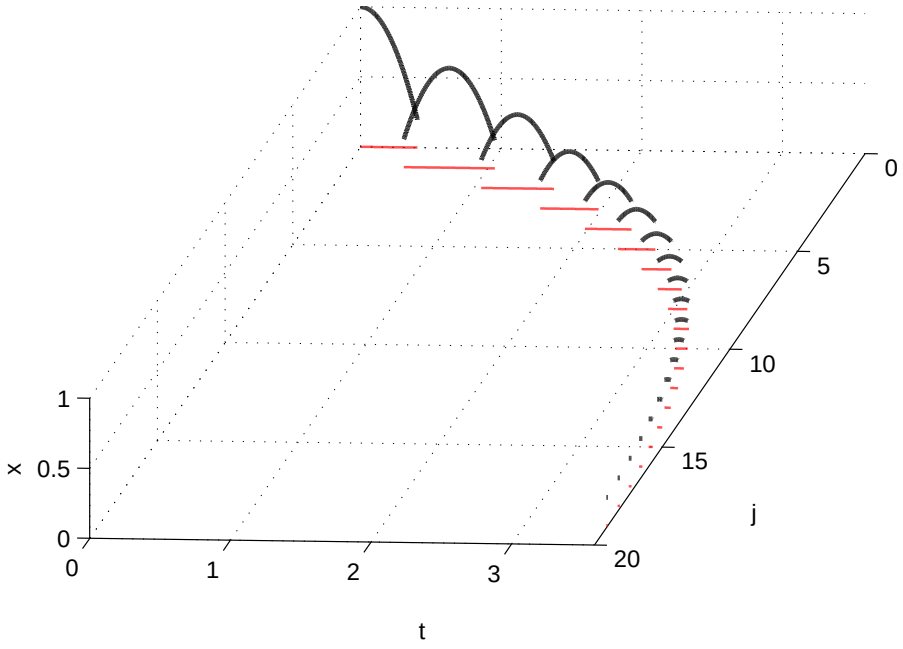

Figure 10: Hybrid arc corresponding to a solution of Example 1.1: height

```
%
%   inside = C(x) returns 0 if outside of C and 1 inside of C
%
%   inside = D(x) returns 0 if outside of D and 1 inside of D
%
%   TSPAN = [TSTART TFINAL] is the time interval. JSPAN = [JSTART JSTOP] is
%       the interval for discrete jumps. The algorithm stop when the first stop
%       condition is reached.
%
%   rule for jumps
%       rule = 1 -> priority for jumps
%       rule = 2 (default) -> priority for flows
%
%   options - options for the solver see odeset f.ex.
%       options = odeset('RelTol',1e-6);
%
%   maxStepCoefficient - set the maximum step length. At each run of the
%       integrator the option 'MaxStep' is set to (time length of last
%       integration)*maxStepCoefficient.
%       Default value = 0.1
%
if ~exist('rule','var')
    rule = 2;
end
if ~exist('options','var')
    options = odeset();
```

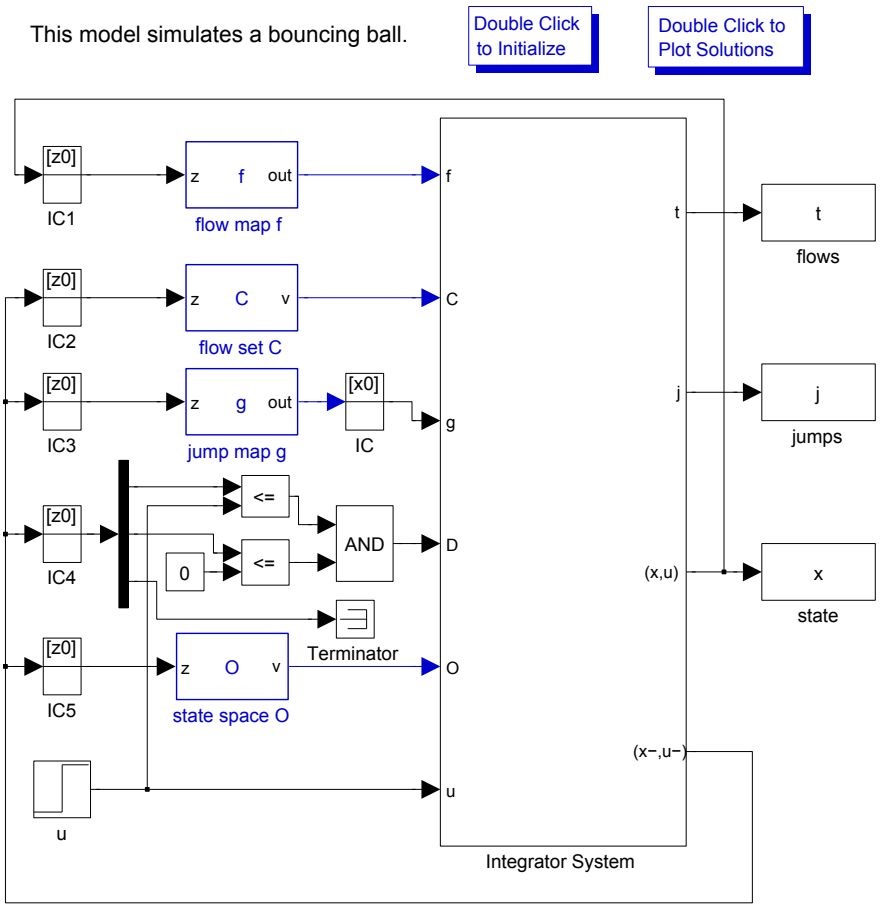

Figure 11: Simulink implementation of bouncing ball example with operator blocks

```
end
if ~exist('maxStepCoefficient','var')
    maxStepCoefficient = .2;
end
% simulation horizon
tstart = TSPAN(1);
tfinal = TSPAN(end);
% simulate
options = odeset(options,'Events',@(t,x) zeroevents(x,C,D,rule));
tout = tstart;
yout = y0.';
jout = JSPAN(1);
j = jout(end);
% Jump if jump is prioritized:
if rule == 1
    while (j<JSPAN(end))
        % Check if value it is possible to jump current position
        insideD = D(yout(end,:).');
        if insideD == 1
            [j tout yout jout] = jump(g,j,tout,yout,jout);
        else
```

```
            break;
        end
    end
end
fprintf('Completed: %3.0f%%',0);
while (j < JSPAN(end) && tout(end) < TSPAN(end))
    % Check if it is possible to flow from current position
    insideC = C(yout(end,:).');
    if insideC == 1
        [t,y] = ode45(@(t,x) f(x),[tout(end) tfinal],yout(end,:).', options);
        nt = length(t);
        tout = [tout; t];
        yout = [yout; y];
        jout = [jout; j*ones(1,nt)'];
        % A good guess of a valid first time step is the length of
        % the last valid time step, so use it for faster computation.
        options = odeset(options,'InitialStep',t(end)-t(nt-1),...
            'MaxStep',(t(end)-t(1))*maxStepCoefficient);
    end
    %Check if it possible to jump
    insideD = D(yout(end,:).');
    if insideD == 0
        break;
    else
        if rule == 1
            while (j<JSPAN(end))
                % Check if value it is possible to jump current position
                insideD = D(yout(end,:).');
                if insideD == 1
                    [j tout yout jout] = jump(g,j,tout,yout,jout);
                else
                    break;
                end
            end
        else
            [j tout yout jout] = jump(g,j,tout,yout,jout);
        end
    end
    fprintf('\b\b\b\b%3.0f%%',100*tout(end)/TSPAN(end));
end
t = tout;
y = yout;
j = jout;
fprintf('\nDone\n');
end
function [value,isterminal,direction] = zeroevents(x,C,D,rule )
isterminal = 1;
direction = -1;
insideC = C(x);
if insideC == 0
    % Outside of C
    value = 0;
elseif (rule == 1)
    % If priority for jump stop if inside D
```

```
    insideD = D(x);
    if insideD == 1
        % Inside D, inside C
        value = 0;
    else
        % outside D, inside C
        value = 1;
    end
else
    % If inside C and not priority for jump or priority of jump and outside
    % of D
    value = 1;
end
end
function [j tout yout jout] = jump(g,j,tout,yout,jout)
% Jump
j = j+1;
y = g(yout(end,:).');
% Save results
tout = [tout; tout(end)];
yout = [yout; y.'];
jout = [jout; j];
end
```

For Matlab/Simulink files of this example, see Examples/Example_1.2b

□

Example 1.3 (vehicle following a track with boundaries)

Consider a vehicle traveling along a given track modeled by a Dubins vehicle model with state x where x is a vector with three components given by $\dot{\xi}_1 = u_1 \cos \xi_3$, $\dot{\xi}_2 = u_1 \sin \xi_3$, and $\dot{\xi}_3 = u_2$. u_1 is the tangential velocity of the vehicle, ξ_1 and ξ_2 describe the vehicle's position, and ξ_3 is the vehicle's orientation angle. Also consider a switching controller attempting to keep the vehicle inside the boundaries of the track while traveling. A state $q \in \{1, 2\}$ is used to define the modes of operation of the controller. The state of the closed-loop system is given by $x := [\xi^\top \ q]^\top$. For the simulation of the described system with a constant input and regular data given by

$$O := \mathbb{R}^3 \times \{1, 2\}, f(x, u) := \begin{bmatrix} \begin{bmatrix} u_1 \cos(\xi_3) \\ u_1 \sin(\xi_3) \\ u_2 \end{bmatrix} \\ 0 \end{bmatrix}, \tag{4}$$

$$C := \left\{(x, u) \in \mathbb{R}^3 \times \{1, 2\} \mid (\xi_1 \leq 1, q = 2) \text{ or } (\xi_1 \geq -1, q = 1)\right\}, \tag{5}$$

$$g(x, u) := \begin{cases} \begin{bmatrix} \xi \\ 2 \end{bmatrix} & \xi_1 \leq -1, q = 1 \\ \begin{bmatrix} \xi \\ 1 \end{bmatrix} & \xi_1 \geq 1, q = 2 \end{cases}, \tag{6}$$

$$D := \left\{(x, u) \in \mathbb{R}^3 \times \{1, 2\} \mid (\xi_1 \geq 1, q = 2) \text{ or } (\xi_1 \leq -1, q = 1)\right\} \tag{7}$$

When $q = 1$, the vehicle is traveling to the left, and when $q = 2$, the vehicle is traveling to the right. The Matlab scripts in each of the function blocks of the implementation above are given as follows. The tangential velocity of the vehicle is chosen to be $u_1 = 1$, and the initial orientation angle is chosen to be $\xi_3 = \pi/4$ radians.

```
function out = f(z)
v = 1; %tangential velocity
```

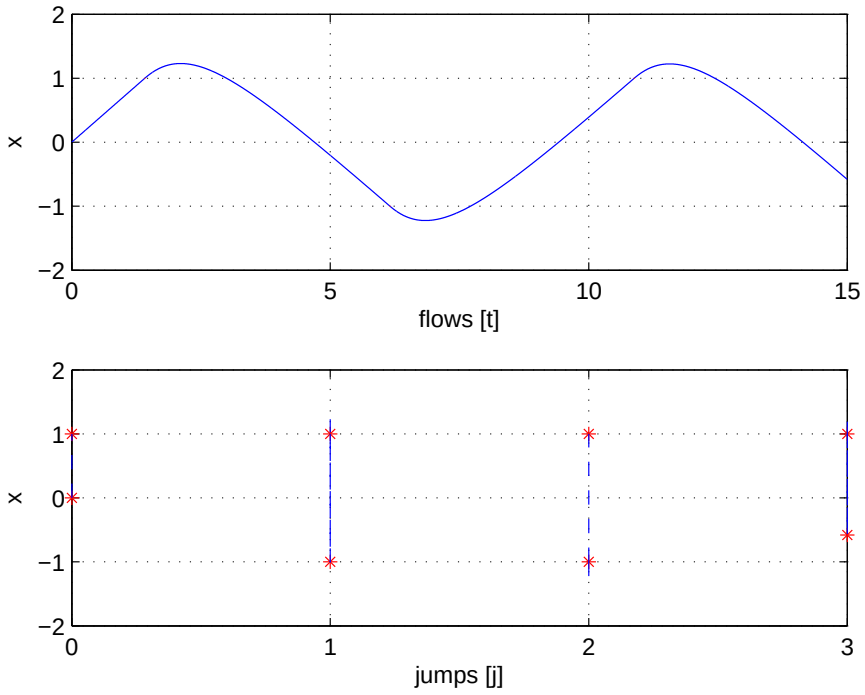

Figure 12: Solution of Example 1.3: trajectory

```
% state
x1 = x(1);      %x-position
x2 = x(2);      %y-position
x3 = x(3);      %orientation angle
q = x(4);
%input
u1 = u(1);
% q = 1 --> going left
% q = 2 --> going right
if q == 1
    r = 3*pi/4;
elseif q == 2
    r = pi/4;
else
    r = 0;
end
% flow map
x1dot = v*cos(x3);  %tangential velocity in x-direction
x2dot = v*sin(x3);  %tangential velocity in y-direction
x3dot = -x3 + r;    %angular velocity
qdot = 0;
out = [x1dot;x2dot;x3dot;qdot];

function [v] = C(z)
% state
x1 = x(1);      %x-position
```

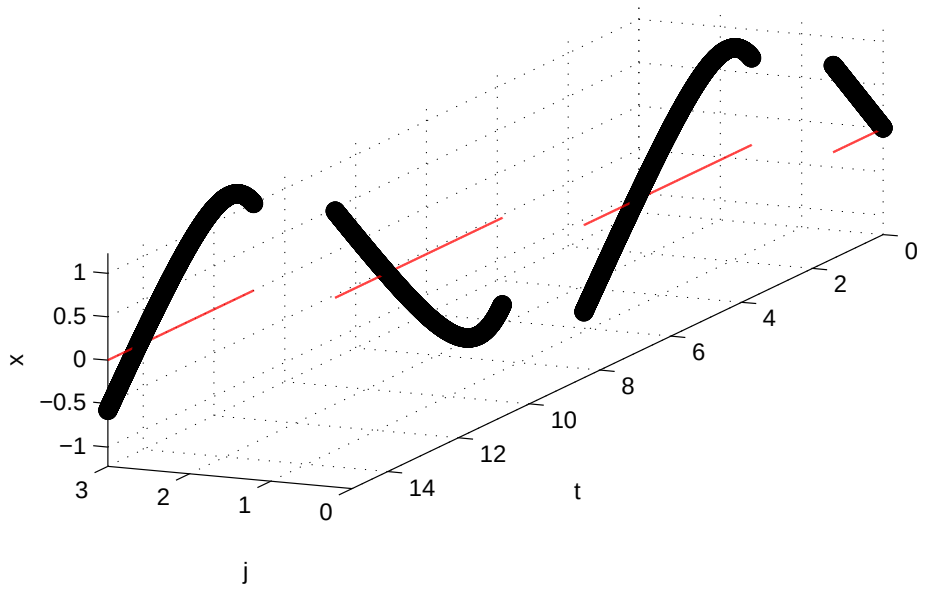

Figure 13: Hybrid arc corresponding to a solution of Example 1.3: trajectory

```
x2 = x(2);      %y-position
x3 = x(3);      %orientation angle
q = x(4);
%input
u1=u(1);
% q = 1 --> going left
% q = 2 --> going right
if (x1 < 1) && (q == 2)  % flow condition
    v = 1;  % report flow
elseif (x1 > -1) && (q == 1) %flow condition
    v = 1;  %report flow
else
    v = 0;   % do not report flow
end

function out = g(z)
% state
x1 = x(1);      %x-position
x2 = x(2);      %y-position
x3 = x(3);      %orientation angle
q = x(4);
%input
u1=u(1);
% q = 1 --> going left
% q = 2 --> going right
x1plus=x1;
```

```
x2plus=x2;
x3plus=x3;
qplus = q;
%jump map
if (x1 >= 1) && (q == 2)
   qplus = 3-q;
elseif (x1 <= -1) && (q == 1)
   qplus = 3-q;
else
   qplus = 0;
end
out = [x1plus;x2plus;x3plus;qplus];
```

```
function [v] = D(z)
% state
x1 = x(1);      %x-position
x2 = x(2);      %y-position
x3 = x(3);      %orientation angle
q = x(4);
%input
u1=u(1);
% q = 1 --> going left
% q = 2 --> going right
 if (x1 >= 1) && (q == 2)  % jump condition
     v = 1;  % report jump
 elseif (x1 <= -1) && (q == 1)  % jump condition
     v = 1;   % report jump
 else
     v = 0; % do not report jump
 end
```

```
function [v] = O(z)
v = 1;  % in the state space
```

A solution to the system of a vehicle following a track in between boundaries at -1 and 1, and with $T = 15, J = 10$, $rule = 1$, is depicted in Figure 12 (trajectory). Both the projection onto t and j are shown. Figure 13 depicts the corresponding hybrid arc.

For Matlab/Simulink files of this example, see Examples/Example_1.3.

□

Example 1.4 (interconnection of hybrid systems $\mathcal{H}1$ (bouncing ball) and $\mathcal{H}2$ (moving platform))

Consider a bouncing ball ($\mathcal{H}1$) bouncing on a platform and a platform ($\mathcal{H}2$) at some initial height and converging to the ground with a height equal to zero. With this interconnection, the bouncing ball will contact the platform, bounce back up, and cause a jump in height of the platform so that it gets closer to the ground. After some time, both the ball and the platform will converge to the ground. In order to model this system, the output of the bouncing ball becomes the input of the moving platform, and vice versa. For the simulation of the described system with regular data where $\mathcal{H}_1$ is given by

$$O_1 := \mathbb{R}^2 \times \mathbb{R}, f_1(\xi, u_1, v_1) := \begin{bmatrix} \xi_2 \\ -\gamma - b\xi_2 + v_{11} \end{bmatrix}, C_1 := \{(\xi, u_1) \mid \xi_1 \geq u_1, u_1 \geq 0\} \tag{8}$$

$$g_1(\xi, u_1, v_1) := \begin{bmatrix} \xi_1 + \alpha_1 \xi_2^2 \\ e_1|\xi_2| + v_{12} \end{bmatrix}, D_1 := \{(\xi, u_1) \mid \xi_1 = u_1, u_1 \geq 0\}, y_1 = h_1(\xi) := \xi_1 \tag{9}$$

where $\gamma, b, \alpha_1 > 0, e_1 \in [0, 1)$, $\xi = [\xi_1 \ \xi_2]^\top$ is the state, $y_1 \in \mathbb{R}$ is the output, $u_1 \in \mathbb{R}$ and $v_1 = [v_{11} \ v_{12}]^\top \in \mathbb{R}^2$ are the inputs, and the hybrid system $\mathcal{H}_2$ is given by

$$O_2 := \mathbb{R}^2 \times \mathbb{R}, f_2(\eta, u_2, v_2) := \begin{bmatrix} \eta_2 \\ -\eta_1 - 2\eta_2 + v_{12} \end{bmatrix}, C_2 := \{(\eta, u_2) \mid \eta_1 \leq u_2, \eta_1 \geq 0\} \tag{10}$$

$$g_2(\eta, u_2, v_2) := \begin{bmatrix} \eta_1 - \alpha_2|\eta_2| \\ -e_2|\eta_2| + v_{22} \end{bmatrix}, D_2 := \{(\eta, u_2) \mid \eta_1 = u_2, \eta_1 \geq 0\}, y_2 = h_2(\eta) := \eta_1 \tag{11}$$

where $\alpha_2 > 0, e_2 \in [0, 1)$, $\eta = [\eta_1\ \eta_2]^\top \in \mathbb{R}^2$ is the state, $y_2 \in \mathbb{R}$ is the output, and $u_2 \in \mathbb{R}$ and $v_2 = [v_{21}\ v_{22}]^\top \in \mathbb{R}^2$ are the inputs.

Therefore, the interconnection may be defined by the input assignment

$$u_1 = y_2, \qquad u_2 = y_1. \tag{12}$$

v_1 and v_2 are included as external inputs in the model in order to simulate the effects of environmental perturbations, such as a wind gust, on the system.

The Matlab scripts in each of the function blocks of the implementation above are given as follows. The constants for the interconnected system are $\gamma = 0.8$, $b = 0.1$, and $\alpha_1, \alpha_2 = 0.1$.

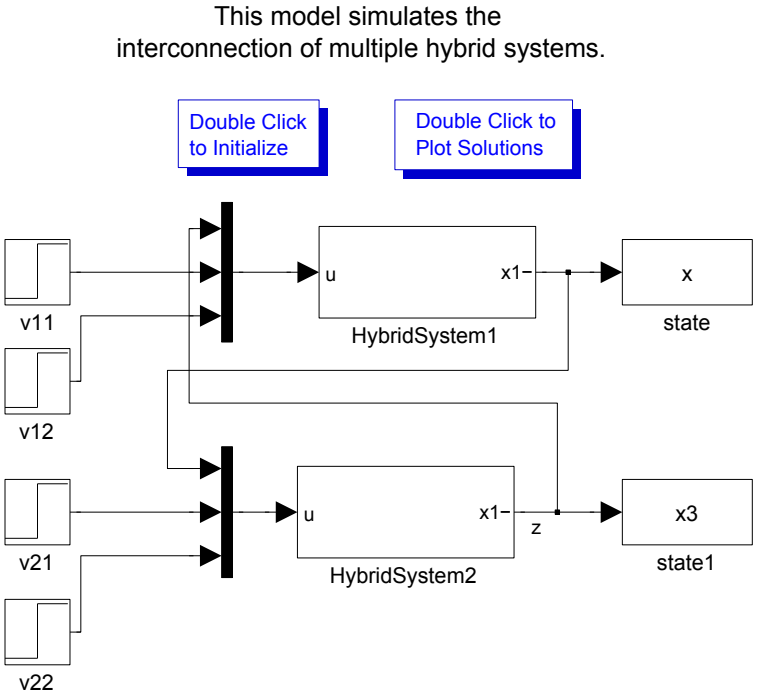

Figure 14: Matlab/Simulink implementation of interconnected hybrid systems $\mathcal{H}1$ and $\mathcal{H}2$

For hybrid system $\mathcal{H}_1$:

```
global n m;
% n = # of state components
% m = # of input components
function out = f(z)
% state
x = z(1:n);
x1 = x(1);
x2 = x(2);
%input
u = z(n+1:n+m);
```

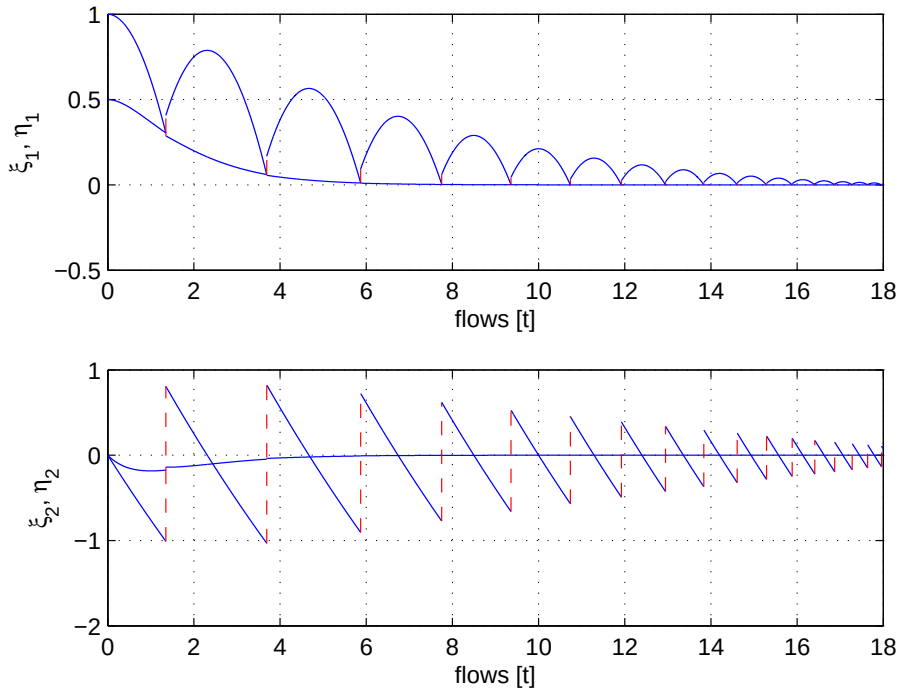

Figure 15: Solution of Example 1.4: height and velocity

```
u1 = u(1);
u2 = u(2);
u3 = u(3);
% flow map
x1dot = x2;
x2dot = -0.8-0.1*x2+u2;
out = [x1dot;x2dot];
```

```
function [v] = C(z)
% state
x = z(1:n);
x1 = x(1);
x2 = x(2);
%input
u = z(n+1:n+m);
u1 = u(1);
u2 = u(2);
u3 = u(3);
if (x1 >= u1)  % flow condition
    v = 1;  % report flow
else
    v = 0;   % do not report flow
end
```

```
function out = g(z)
% state
x = z(1:n);
```

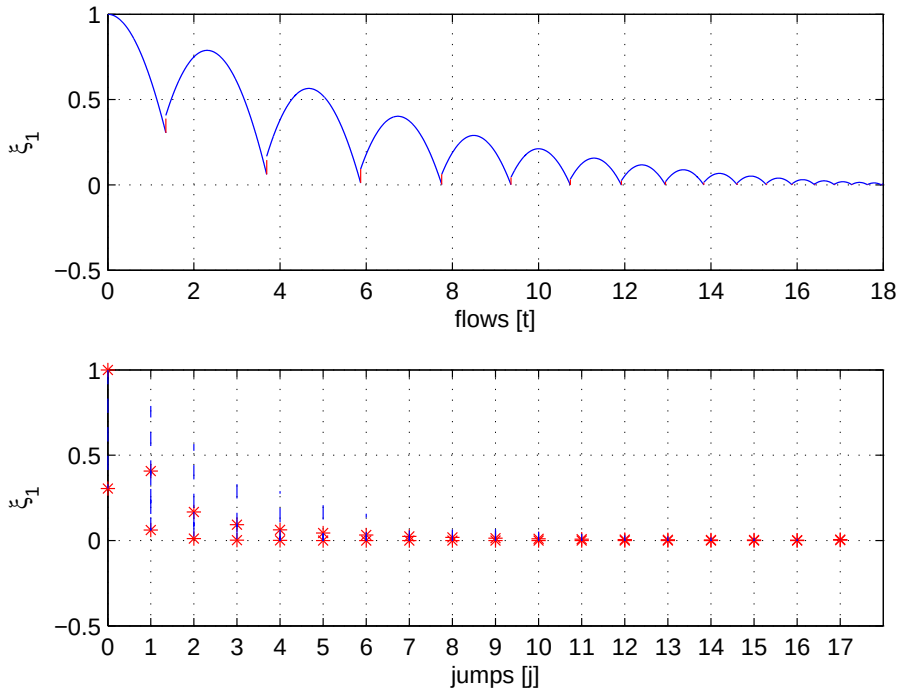

Figure 16: Solution of Example 1.4 for system $\mathcal{H}1$: height

```
x1 = x(1);
x2 = x(2);
%input
u = z(n+1:n+m);
u1 = u(1);
u2 = u(2);
u3 = u(3);
%jump map
x1plus=u1+0.1*x2^2;
x2plus=0.8*abs(x2)+u3;
out = [x1plus;x2plus];

function [v] = D(z)
% state
x = z(1:n);
x1 = x(1);
x2 = x(2);
%input
u = z(n+1:n+m);
u1 = u(1);
u2 = u(2);
u3 = u(3);
if (x1 <= u1) % jump condition
    v = 1;  % report jump
else
    v = 0;   % do not report jump
```

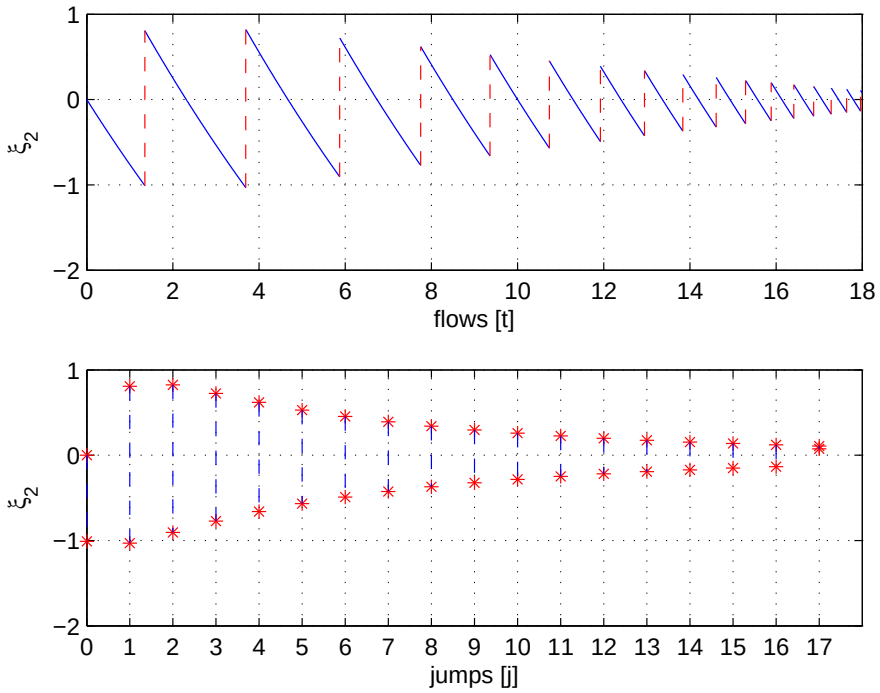

Figure 17: Solution of Example 1.4 for system $\mathcal{H}1$: velocity

```
end

function [v] = O(u)
v = 1;  % in the state space
```

For hybrid system $\mathcal{H}_2$:

```
function out = f(z)
% state
x = z(1:n);
x1 = x(1);
x2 = x(2);
%input
u = z(n+1:n+m);
u1 = u(1);
u2 = u(2);
u3 = u(3);
% flow map
x1dot = x2;
x2dot = -x1-2*x2+u2;
out = [x1dot;x2dot];

function [v] = C(z)
% state
x = z(1:n);
x1 = x(1);
x2 = x(2);
```

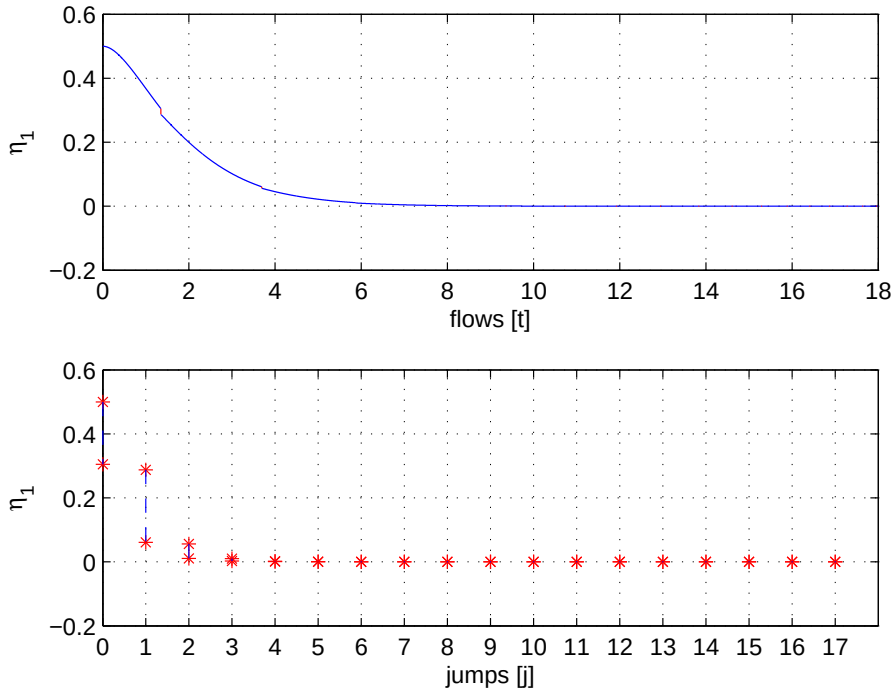

Figure 18: Solution of Example 1.4 for system $\mathcal{H}2$: height

```
%input
u = z(n+1:n+m);
u1 = u(1);
u2 = u(2);
u3 = u(3);
if (x1 <= u1)  % flow condition
    v = 1;  % report flow
else
    v = 0;   % do not report flow
end

function out = g(z)
% state
x = z(1:n);
x1 = x(1);
x2 = x(2);
%input
u = z(n+1:n+m);
u1 = u(1);
u2 = u(2);
u3 = u(3);
% jump map
x1plus = u1-0.1*abs(x2);
x2plus = -0.8*abs(x2)+u3;
out = [x1plus;x2plus];

function [v] = D(z)
```

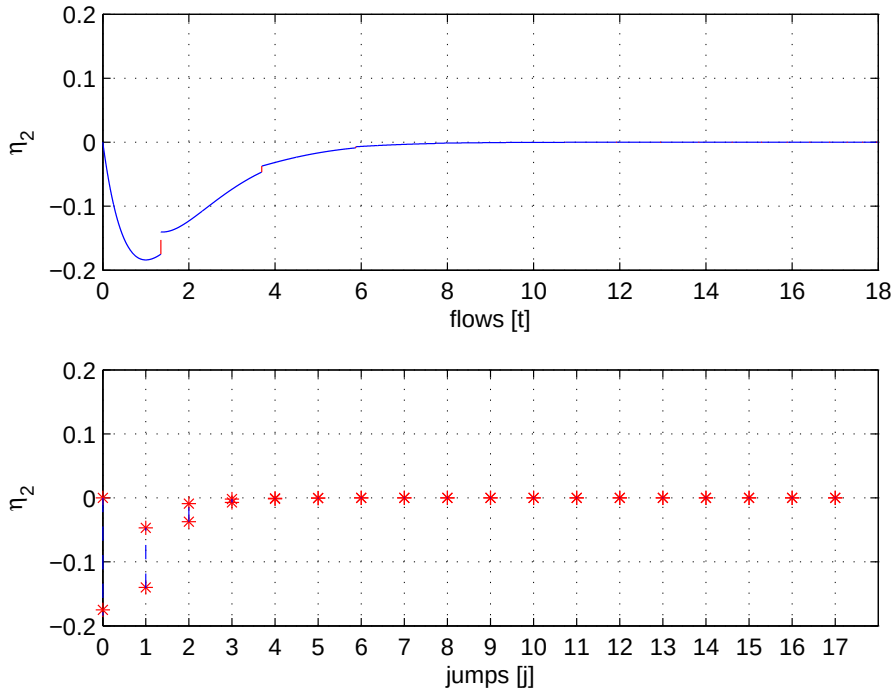

Figure 19: Solution of Example 1.4 for system $\mathcal{H}2$: velocity

```
% state
x = z(1:n);
x1 = x(1);
x2 = x(2);
%input
u = z(n+1:n+m);
u1 = u(1);
u2 = u(2);
u3 = u(3);
if (x1 >= u1) % jump condition
    v = 1;  % report jump
else
    v = 0;   % do not report jump
end

function [v] = O(u)
v = 1;  % in the state space
```

A solution to the interconnection of hybrid systems $\mathcal{H}1$ and $\mathcal{H}2$ with $T = 18$, $J = 20$, $rule = 1$, is depicted in Figure 15. Both the projection onto t and j are shown. A solution to the hybrid system $\mathcal{H}1$ is depicted in Figure 16 (height) and Figure 17 (velocity). A solution to the hybrid system $\mathcal{H}2$ is depicted in Figure 18 (height) and Figure 19 (velocity).

These simulations reflect the expected behavior of the interconnected hybrid systems. Note that in order to implement these systems without premature stopping of the simulation, ξ_1 in g_1 and η_1 in g_2 can be changed to u_1 and u_2, respectively so that $\xi_1^+ = u_1$ and $\eta_1^+ = u_2$.

For Matlab/Simulink files of this example, see Examples/Example_1.4.

□

Example 1.5 (biological example: synchronization of two fireflies)

Consider a biological example of the synchronization of two fireflies flashing. The fireflies can be modeled mathematically as periodic oscillators which tend to synchronize their flashing until they are flashing in phase with each other. The synchronization of the fireflies can be modeled as an interconnection of two hybrid systems where each firefly can be modeled as a hybrid system given by

$$O_i := \mathbb{R}^2, f_i(\tau_i, u_i) := 1, \tag{13}$$

$$C_i := \left\{(\tau_i, u_i) \in \mathbb{R}^2 \mid 0 < \tau_i < 1\right\} \cap \left\{(\tau_i, u_i) \in \mathbb{R}^2 \mid 0 < u_i \leq 1\right\} \tag{14}$$

$$g_i(\tau_i, u_i) := \begin{cases} (1+\varepsilon)\tau_i & (1+\varepsilon)\tau_i < 1 \\ 0 & (1+\varepsilon)\tau_i \geq 1 \end{cases} \tag{15}$$

$$D_i := \left\{(\tau_i, u_i) \in \mathbb{R}^2 \mid \tau_i = 1\right\} \cup \left\{(\tau_i, u_i) \in \mathbb{R}^2 \mid u_i = 1\right\}. \tag{16}$$

A state value of $\tau_i = 1$ corresponds to a flash, and after each flash, the firefly automatically resets its periodic cycle to $\tau_i = 0$. When one of the fireflies flashes, the other tries to synchronize its flash by jumping ahead in its periodic cycle. This behavior is captured by the biological coefficient, ε.

The interconnection diagram for this example is simpler than in the previous example because now no external inputs are being considered. The only event that affects the flashing of a firefly is the flashing of the other firefly. The interconnection diagram can be seen in Figure 20.

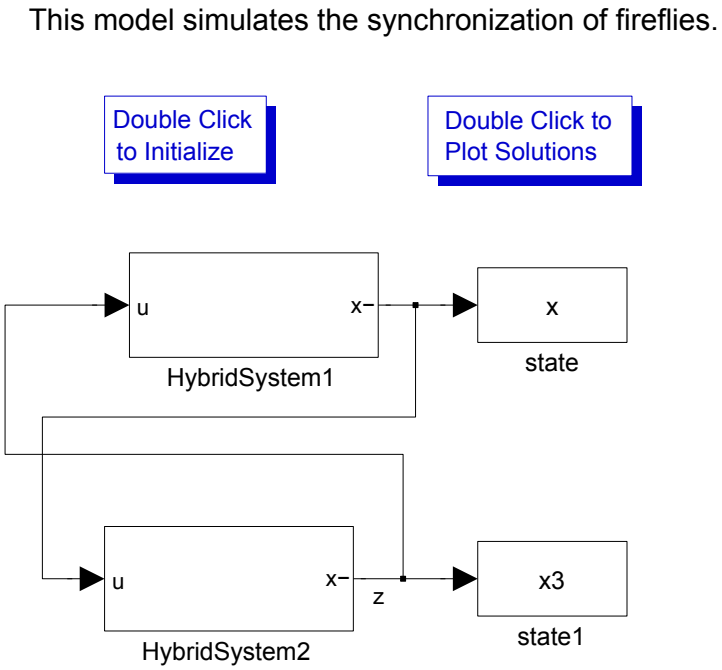

Figure 20: Interconnection Diagram for Example 1.5

For hybrid system $\mathcal{H}_i$:

```
function out = f(z)
% state
x = z(1:n);
%input
u = z(n+1:n+m);
% flow map
```

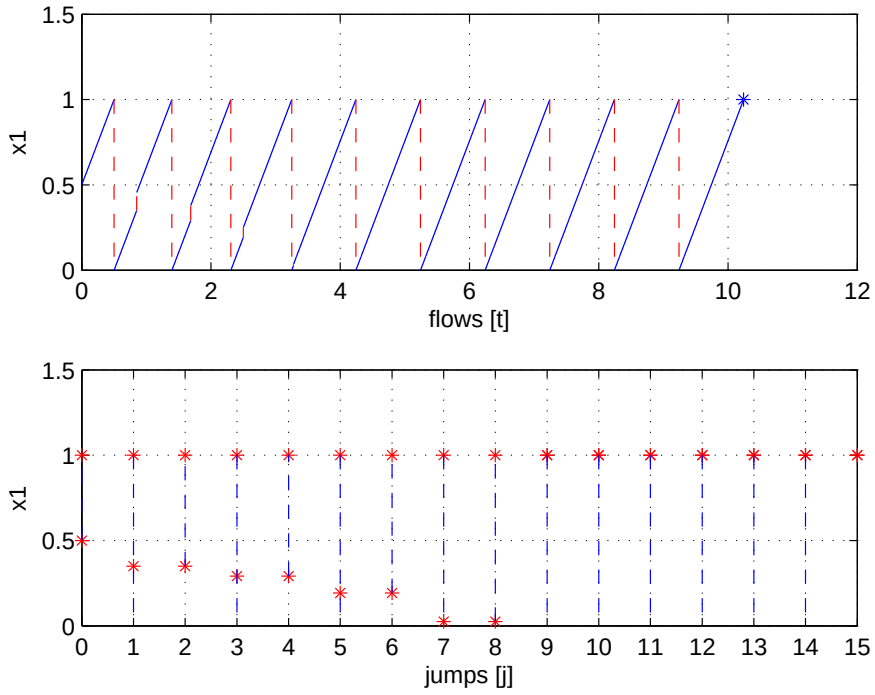

Figure 21: Solution of Example 1.5 for system $\mathcal{H}1$

```
xdot = 1;
out = xdot;

function v  = C(z)
% state
x = z(1:n);
%input
u = z(n+1:n+m);
if (0 < x < 1)  % flow condition
    v = 1;  % report flow
elseif (0 < u <= 1)
    v = 1;   % report flow
else
    v = 0;  % do not report flow
end

function out = g(z)
% state
x = z(1:n);
%input
u = z(n+1:n+m);
% jump map
if (1+e)*x < 1
    xplus = (1+e)*x;
elseif x == 1
    xplus = 0;
elseif (1+e)*x >= 1
```

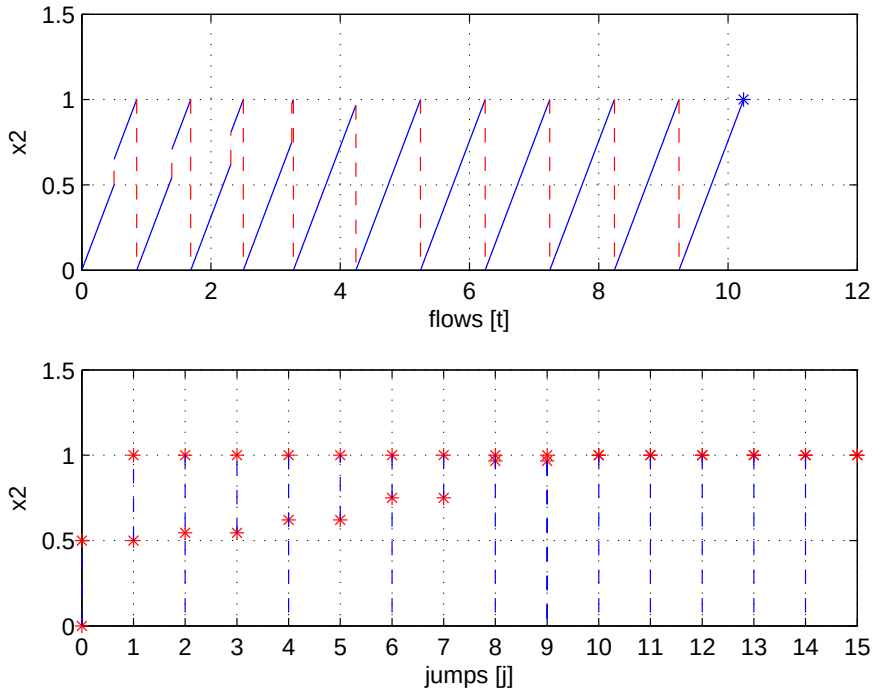

Figure 22: Solution of Example 1.5 for system $\mathcal{H}2$

```
    xplus = 0;
else
    xplus = 0;
end
out = xplus;

function v  = D(z)
% state
x = z(1:n);
%input
u = z(n+1:n+m);
if (u >= 1) % jump condition
    v = 1;  % report jump
elseif (x >= 1) % jump condition
    v = 1;  % report jump
else
    v = 0;  % do not report jump
end

function v  = O(z)
v = 1;  % in the state space
```

A solution to the interconnection of hybrid systems $\mathcal{H}1$ and $\mathcal{H}2$ with $T = 15, J = 15$, $rule = 1$, $\varepsilon = 0.3$ is depicted in Figure 23. Both the projection onto t and j are shown. A solution to the hybrid system $\mathcal{H}1$ is depicted in Figure 21. A solution to the hybrid system $\mathcal{H}2$ is depicted in Figure 22.

These simulations reflect the expected behavior of the interconnected hybrid systems. The fireflies initially flash out of phase with one another and then synchronize to flash in the same phase.

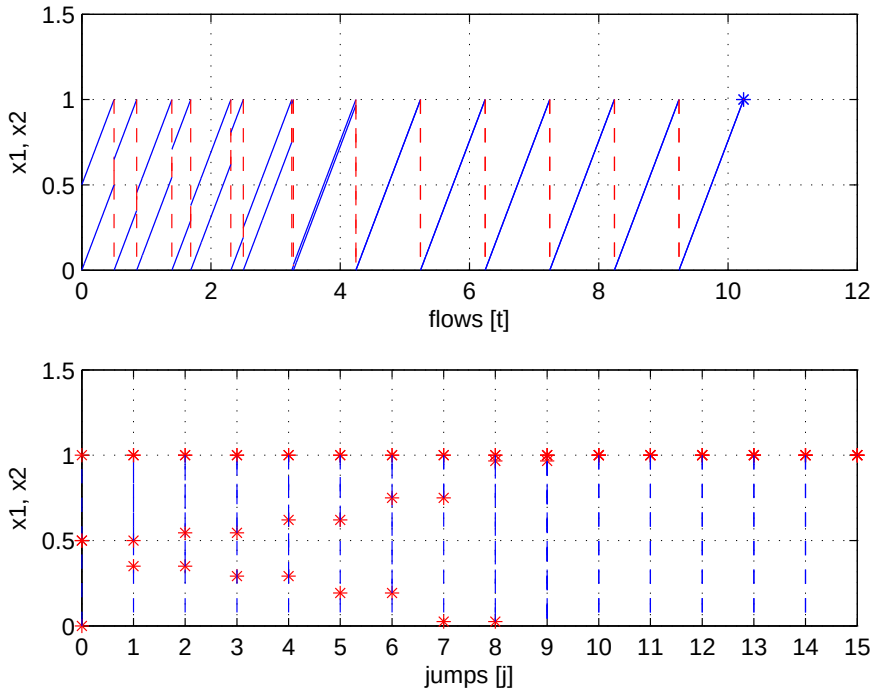

Figure 23: Solution of Example 1.5 for interconnection of $\mathcal{H}1$ and $\mathcal{H}2$

For Matlab/Simulink files of this example, see Examples/Example_1.5.

□

Example 1.6 (analysis of interconnection of two hybrid systems)
Consider two interconnected hybrid systems $\mathcal{H}_i$, $i \in \{1, 2\}$, given by

$$\mathcal{H}_i \left\{ \begin{array}{rcll} \dot{x_i} & = & f_i(x_i, \tilde{u}_i, u_i) := -a_i x_i + b_i \tilde{u}_i + u_i & (x_i, \tilde{u}_i, u_i) \in C_i \\ x_i^+ & = & g_i(x_i, \tilde{u}_i, u_i) := \tilde{u}_i & (x_i, \tilde{u}_i, u_i) \in D_i \\ y_i & = & h_i(x_i) := x_i, & \end{array} \right.$$

where

$C_i := \{(x_i, \tilde{u}_i, u_i) \ : \ \tilde{u}_i(x_i - \varepsilon_i \tilde{u}_i) \leq 0\}$, $D_i := \{(x_i, \tilde{u}_i, u_i) \ : \ \tilde{u}_i(x_i - \varepsilon_i \tilde{u}_i) \geq 0\}, a_i, b_i, \varepsilon_i > 0$ and $x_i, \tilde{u}_i, u_i \in \mathbb{R}$.

For motivation and analysis of this example, the reader is referred to [4].

Figures 24, 25, and 27 show the solution to this example when $a_1 = 10$, $a_2 = 30$, $b_1 = .5$, $b_2 = 1$, $\varepsilon_1 = 2$, $\varepsilon_2 = 3$, and initial conditions x_{10}= x_{20}= 5. The Matlab scripts in each of the function blocks of the implementation above are given as follows.

For hybrid system $\mathcal{H}_1$:

```
function out = f(z)
% state
x = z(statevect);
x1 = x(1);
%input
u = z(inputvect);
```

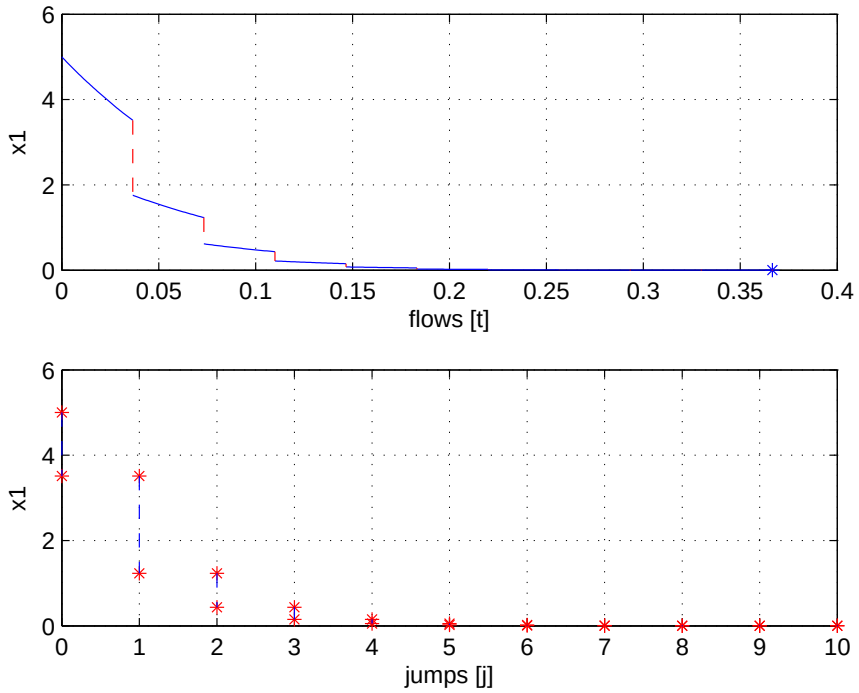

Figure 24: Solution of Example 1.6 for system $\mathcal{H}1$

```
u1 = u(1);
u2 = u(2);
u3 = u(3);
% flow map
x1dot = -a1*x1+b1*u1+u2;
out = [x1dot];

function v  = C(z)
% state
x = z(statevect);
x1 = x(1);
%input
u = z(inputvect);
u1 = u(1);
u2 = u(2);
u3 = u(3);
%flow set
if (u1*(x1-e1*u1) <= 0)  % flow condition
    v = 1;  % report flow
else
    v = 0;   % do not report flow
end

function out = g(z)
% state
x = z(statevect);
x1 = x(1);
```

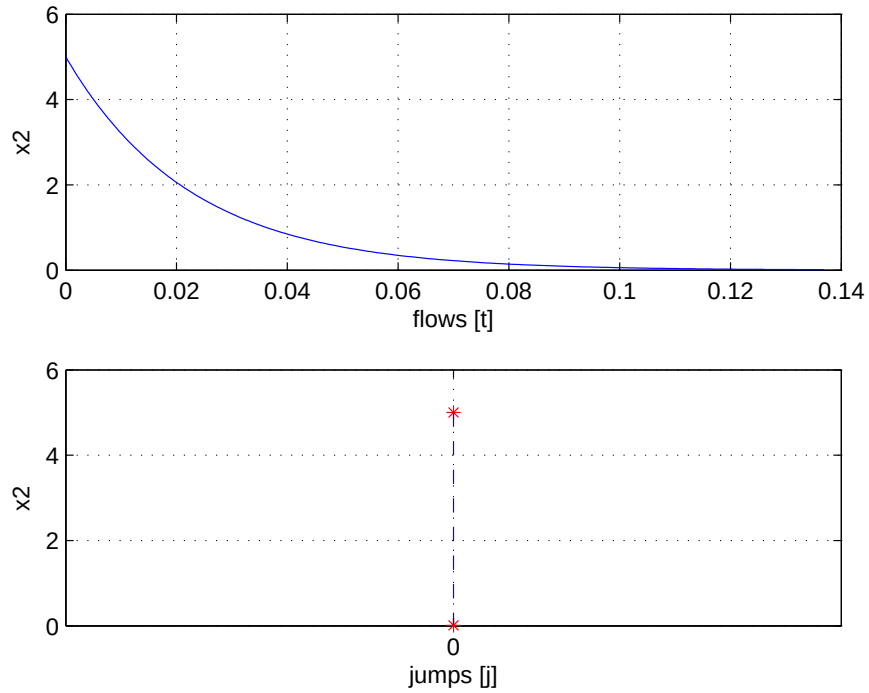

Figure 25: Solution of Example 1.6 for system $\mathcal{H}2$

```
%input
u = z(inputvect);
u1 = u(1);
u2 = u(2);
u3 = u(3);
%jump map
x1plus = u1;
out = [x1plus];

function v  = D(z)
% state
x = z(statevect);
x1 = x(1);
%input
u = z(inputvect);
u1 = u(1);
u2 = u(2);
u3 = u(3);
%jump set
if (u1*(x1-e1*u1) >= 0) % jump condition
    v = 1;  % report jump
else
    v = 0;   % do not report jump
end

function v = O(z)
v = 1;  % in the state space
```

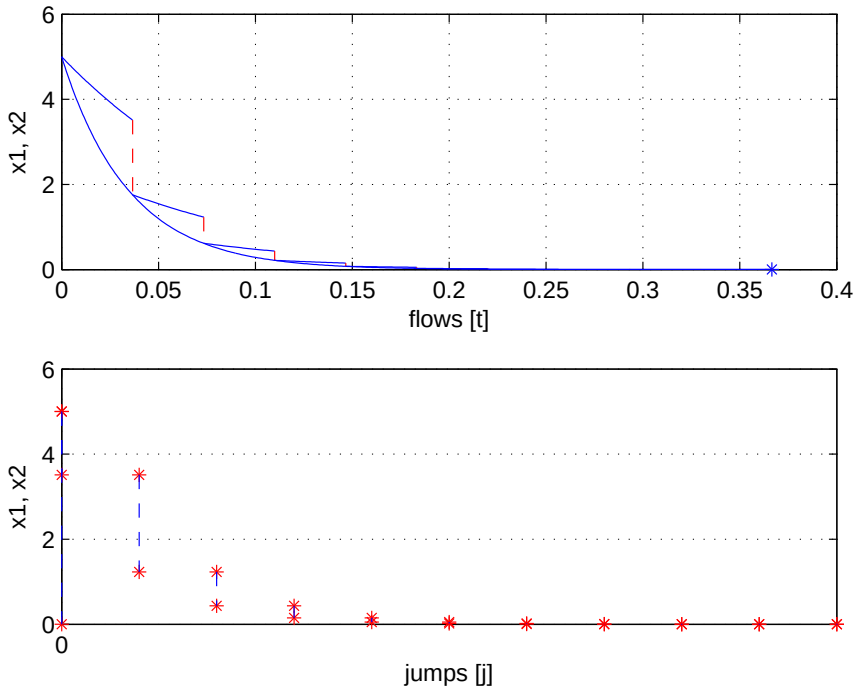

Figure 26: Solution of Example 1.6 for interconnection of $\mathcal{H}1$ and $\mathcal{H}2$

For hybrid system $\mathcal{H}_2$:

```
function out = f(z)
% state
x = z(statevect);
x1 = x(1);
%input
u = z(inputvect);
u1 = u(1);
u2 = u(2);
u3 = u(3);
% flow map
x1dot = -a2*x1+b2*u1+u2;
out = [x1dot];

function v  = C(z)
% state
x = z(statevect);
x1 = x(1);
%input
u = z(inputvect);
u1 = u(1);
u2 = u(2);
u3 = u(3);
%flow set
if (u1*(x1-e2*u1) <= 0)  % flow condition
    v = 1;  % report flow
```

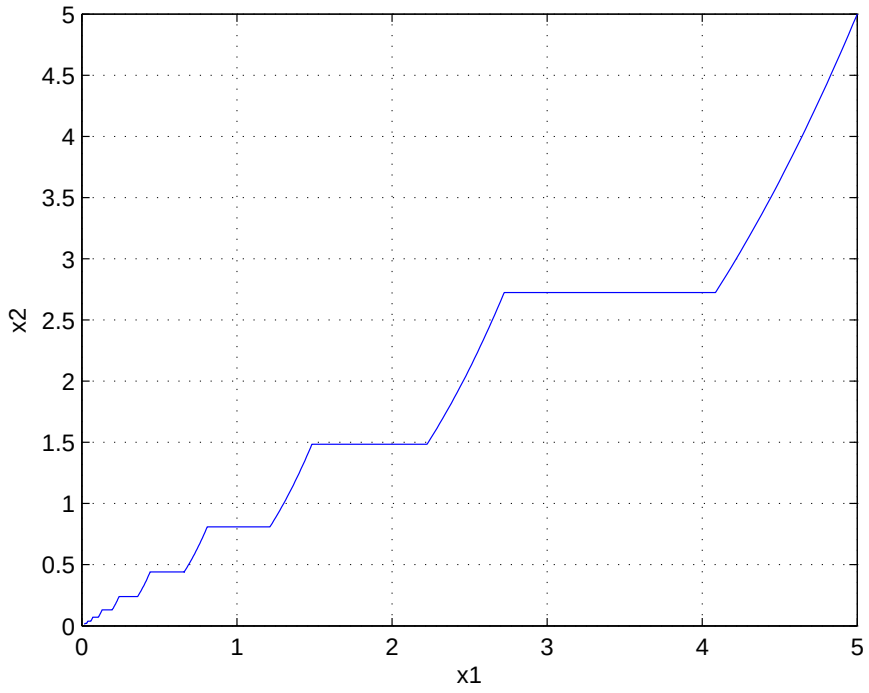

Figure 27: Solution of Example 1.6: state of $\mathcal{H}1$, x_1, vs. state of $\mathcal{H}2$, x_2

```
else
    v = 0;   % do not report flow
end

function out = g(z)
% state
x = z(statevect);
x1 = x(1);
%input
u = z(inputvect);
u1 = u(1);
u2 = u(2);
u3 = u(3);
% jump map
x1plus = u1;
out = [x1plus];

function v  = D(z)
% state
x = z(statevect);
x1 = x(1);
%input
u = z(inputvect);
u1 = u(1);
u2 = u(2);
u3 = u(3);
%jump set
```

```
if (u1*(x1-e2*u1) >= 0) % jump condition
    v = 1;  % report jump
else
    v = 0;   % do not report jump
end

function v = O(z)
v = 1;  % in the state space
```

These simulations reflect the expected behavior the the interconnected hybrid systems. Specifically note that the origin of the systems are stable with the chosen values for a_1, a_2, b_1, and b_2. This can be seen in Figure 27 as the solution converges to zero from the initial condition of 5.

For Matlab/Simulink files of this example, see Examples/Example_1.6.

□

Example 1.7 (a simple mathematical example to show different type of simulation results)

Consider the hybrid system with data

$$O := \mathbb{R},\ f(x) := -x,\ C := [0,1],\ g(x) := 1 + \mathrm{mod}(x,2),\ D := \{1\} \cup \{2\}\ .$$

Note that solutions from $\xi = 1$ and $\xi = 2$ are nonunique. The following simulations show the use of the variable *rule* in the *Jump Logic block*.

Jumps enforced:

A solution from $x0 = 1$ with $T = 10, J = 20$, $rule = 1$ is depicted in Figure 28. The solution jumps from 1 to 2, and from 2 to 1 repetitively.

Flows enforced:

A solution from $x0 = 1$ with $T = 10, J = 20$, $rule = 2$ is depicted in Figure 29. The solution flows for all time and converges exponentially to zero.

Random rule:

A solution from $x0 = 1$ with $T = 10, J = 20$, $rule = 3$ is depicted in Figure 30. The solution jumps to 2, then jumps to 1 and flows for the rest of the time converging to zero exponentially.

Enlarging D to

$$D := [1/50, 1] \cup \{2\}$$

causes the overlap between C and D to be "thicker". The simulation result is depicted in Figure 31 with the same parameters used in the simulation in Figure 30. The plot suggests that the solution jumps several times until $x < 1/50$ from where it flows to zero. However, Figure 32, a zoomed version of Figure 31, shows that initially the solution flows and that at $(t, j) = (0.2e - 3, 0)$ it jumps. After the jump, it continues flowing, then it jumps a few times, then it flows, etc. The combination of flowing and jumping occurs while the solution is in the intersection of C and D, where the selection of whether flowing or jumping is done randomly due to using $rule = 3$.

This simulation also reveals that this implementation does not precisely generate hybrid arcs. The maximum step size was set to $0.1e - 3$. The solution flows during the first two steps of the integration of the flows with maximum step size. The value at $t = 0.1e - 3$ is very close to 1. At $t = 0.2e - 3$, instead of assuming a value given by the flow map, the value of the solution is about 0.5, which is the result of the jump occurring at $(0.2e - 3, 0)$. This is the value stored in x at such time by the integrator. Note that the value of x' at $(0.2e - 3, 0)$ is the one given by the flow map that triggers the jump, and if available for recording, it should be stored in $(0.2e - 3, 0)$. This is a limitation of the current implementation.

□

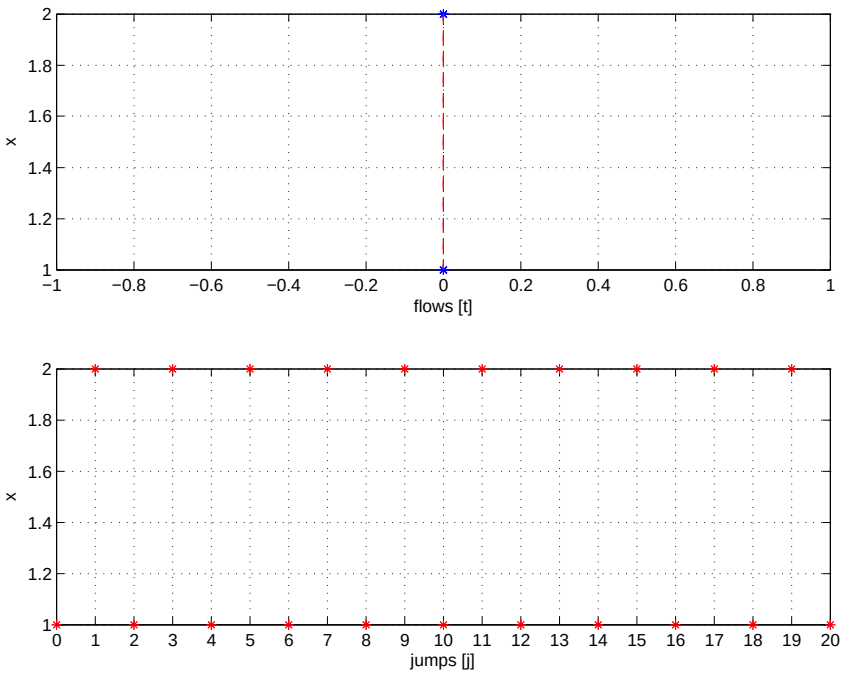

Figure 28: Solution of Example 1.7 with forced jumps logic.

The following simulations show the *Stop Logic block* stopping the simulation at different events.

Solution outside O:

Replacing O by $(-1, 2)$, a solution starting from $x0 = 1$ with $T = 10, J = 20$, $rule = 1$ fails to exists after the first jump. This is depicted in Figure 33 (cf. Figure 28)

Solution outside $C \cup D$:

The same behavior as the one just outlined arises with $O = \mathbb{R}$ but with $D = \{1\}$. The simulation stops since the solution leaves $C \cup D$.

Solution reaches the boundary of C from where jumps are not possible:

Finally, taking $O = \mathbb{R}$ and replacing the flow set by $[1/2, 1]$ a solution starting from $x0 = 1$ with $T = 10, J = 20$ and $rule = 2$ flows for all time until it reaches the boundary of C where jumps are not possible. Figure 34 shows this.

Note that in this implementation, the Stop Logic is such that when the state of the hybrid system is not in $(C \cup D) \cap O$, then the simulation is stopped. In particular, if this condition becomes true while flowing, then the last value of the computed solution will not belong to either C or O, or both, depending on the situation. It could be desired to be able to recompute the solution so that its last point belongs to the corresponding set. From that point, it should be the case that solutions cannot be continued.

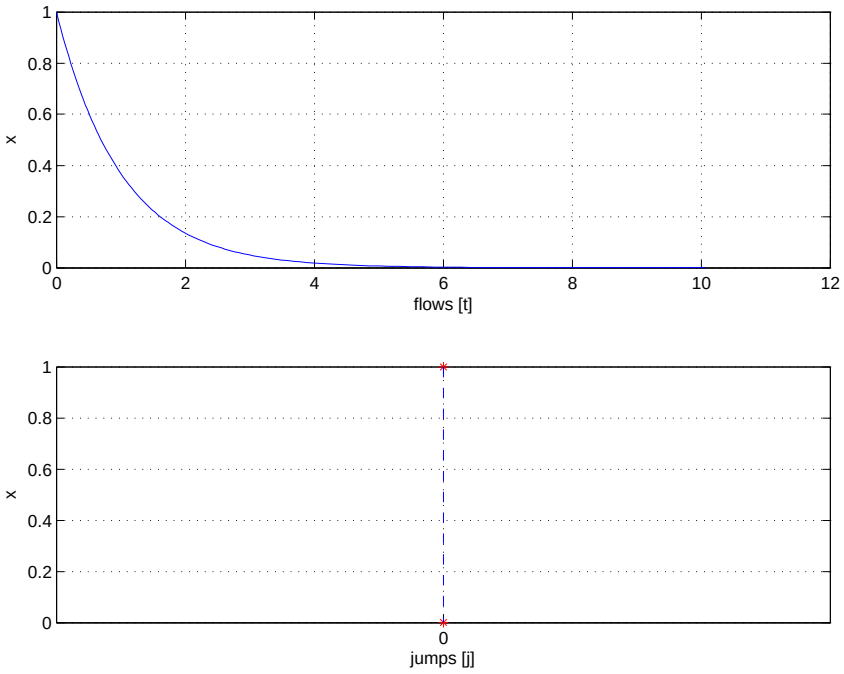

Figure 29: Solution of Example 1.7 with forced flows logic.

For Matlab/Simulink files of this example, see Examples/Example_1.7.

4 Notes

Matlab/Simulink files corresponding to the simulation technique described in this paper can be found at Matlab Central and at the author's website

http://www.u.arizona.edu/∼sricardo/.

5 Acknowledgments

We would like to thank Giampiero Campa for his thoughtful feedback and advice as well as Torstein Ingebrigtsen Bo for his comments and lite simulator code.

6 References

[1] David A. Copp and Ricardo G. Sanfelice, Simulating Hybrid Systems in Matlab/Simulink, v0.4. Hybrid Dynamics and Control Laboratory, University of Arizona.
[2] http://control.ee.ethz.ch/ ifaatic/ex/example1.m. Institut für Automatik - Automatic Control Laboratory, ETH Zurich, 2011.
[3] R. Goebel, R. G. Sanfelice, and A. R. Teel, Hybrid dynamical systems. IEEE Control Systems Magazine, 28-93, 2009.
[4] R. G. Sanfelice and A. R. Teel, Dynamical Properties of Hybrid Systems Simulators. Automatica, 46, No. 2, 239–248, 2010.

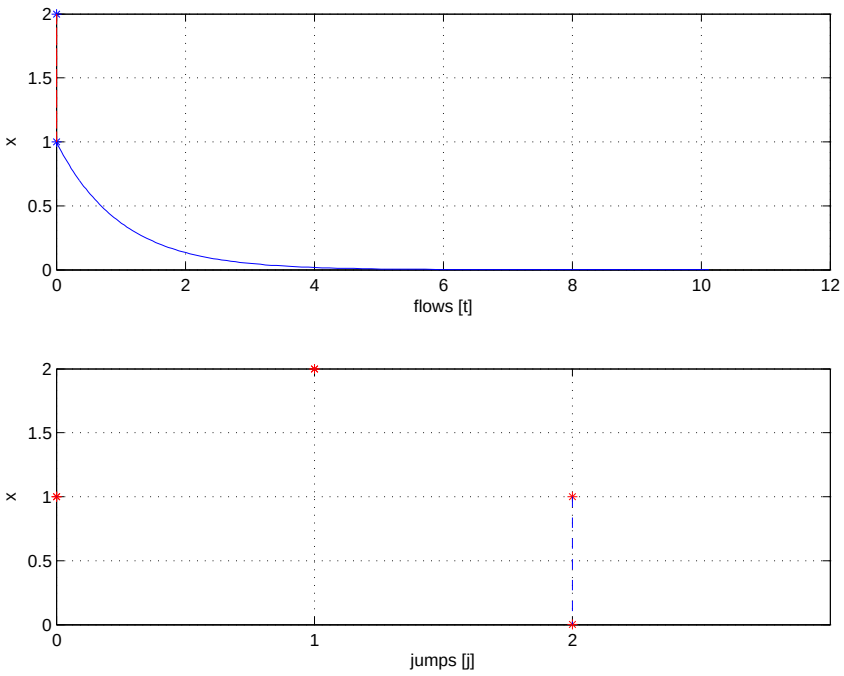

Figure 30: Solution of Example 1.7 with random logic for flowing/jumping.

[5] Ricardo G. Sanfelice, Interconnections of Hybrid Systems: Some Challenges and Recent Results. Submitted, 2011.

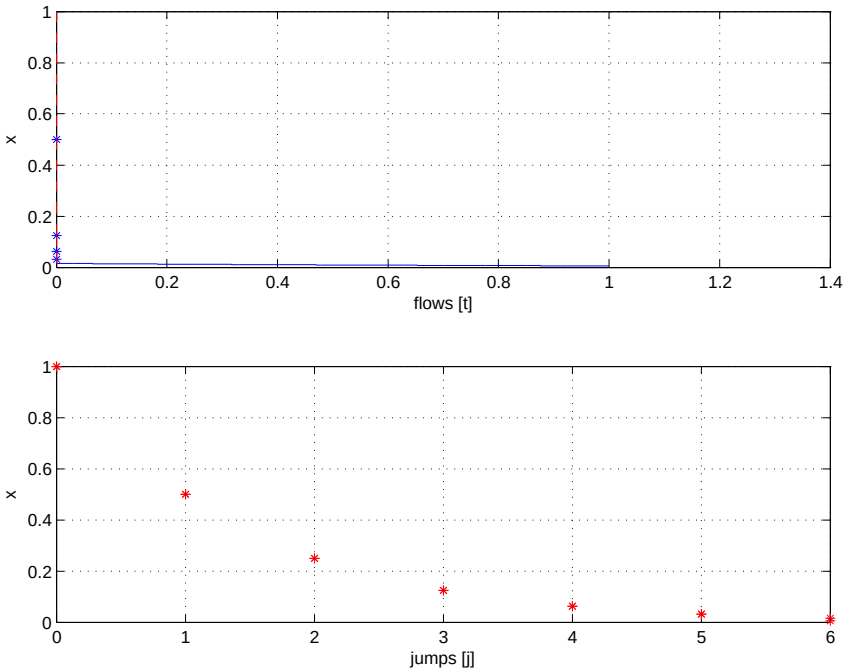

Figure 31: Solution of Example 1.7 with random logic for flowing/jumping.

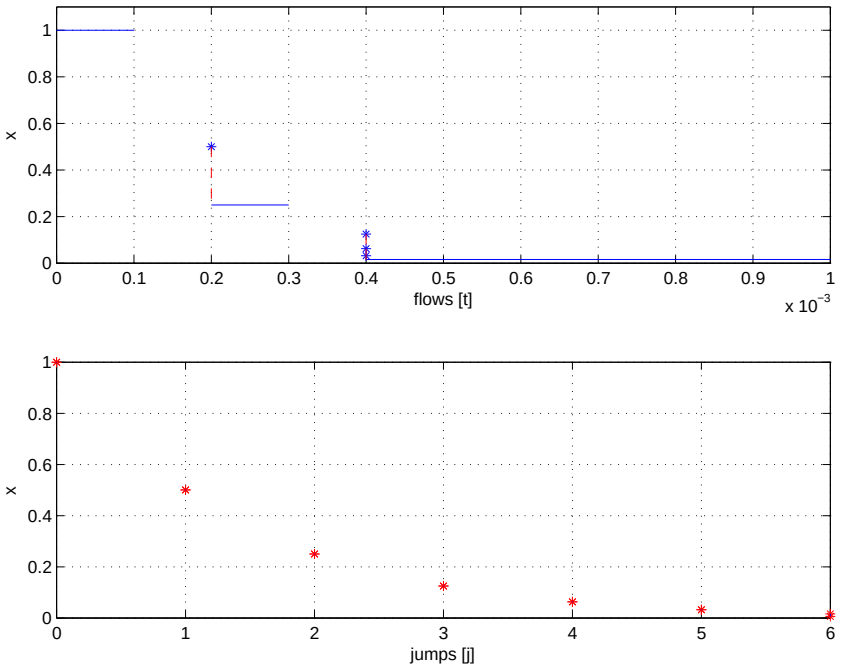

Figure 32: Solution of Example 1.7 with random logic for flowing/jumping. Zoomed version.

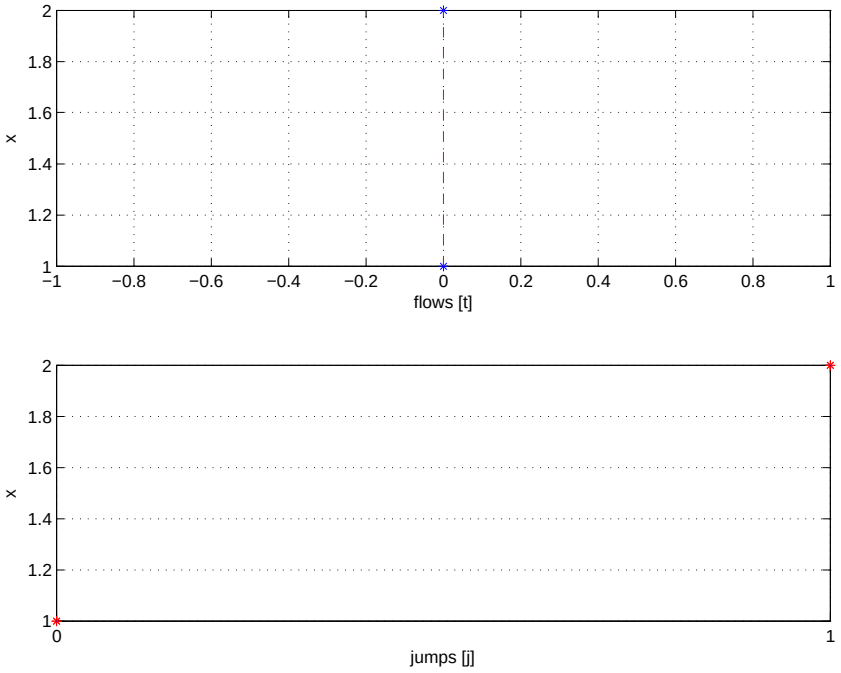

Figure 33: Solution of Example 1.7 with forced jump logic and different O.

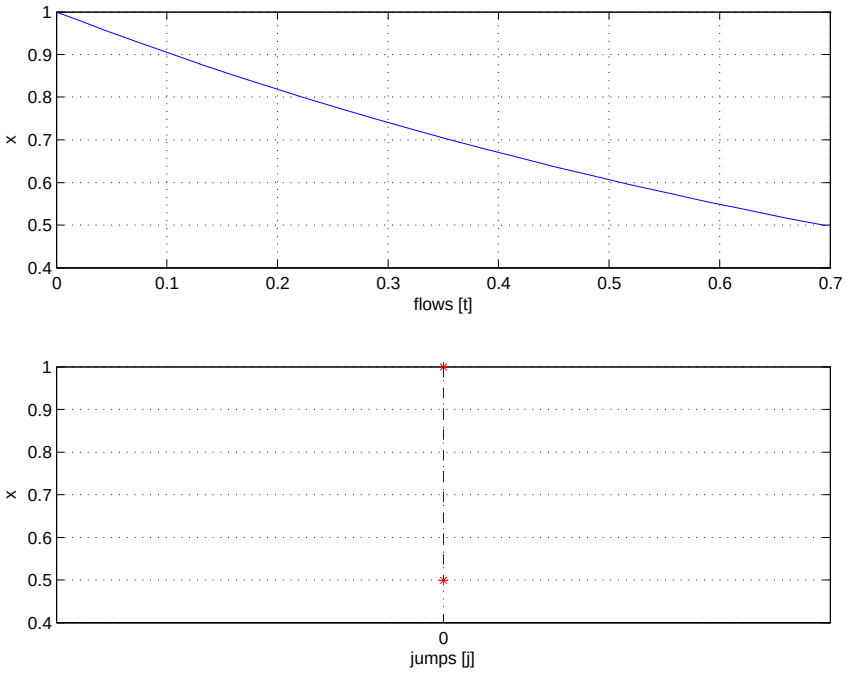

Figure 34: Solution of Example 1.7 with forced flow logic.