ARGESIM Benchmark C7 'Constrained Pendulum' - Solution in MATLAB Environment and Extensions with Linear Approach, Symbolic Approach, Sensitivity, and Integration into TU Vienna's MMT E-Learning Environment

Marko Grujic[2], Jakob Haupt[2], Ypti Hossain[2], Lorenz Klimon[2], Paul Setinek[1], Felix Breitenecker[1*]

[1]Institute of Analysis and Scientific Computing, [2]Inst. of Mechanics and Mechatronics, TU Wien, Wiedner Hauptstrasse 8-10, 1040 Vienna, Austria; **felix.breitenecker@tuwien.ac.at*

SNE 31(4), 2021, 239-254, DOI: 10.11128/sne.31.bne07.10589
Received: 2020-12-10; Revised: 2021-07-05;
Revised: 2021-09-10; Accepted: 2021-09-15
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna
ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

Abstract. The ARGESIM Benchmark 'C7 Constrained Pendulum' is based on the dynamics of a pendulum which hits a pin: hit and release of the pin is a state event, which has to be managed properly. This Educational Benchmark Note, a detailed Benchmark Study, presents four issues for this benchmark. First, the study describes classical approaches, implementation and results for the requested benchmark tasks in MATLAB, Simulink and Stateflow, putting emphasis on the quality of event finding. Second, the study investigates in detail the possibilities of the linear pendulum model for event management: ODE approach, state space approach with exponential matrix, approach with analytical solution, and approach with symbolic computation. Third, the study sketches sensitivity analysis for the model, and fourth, the study presents the implementation of the model into TU Vienna's MMT E-Learning Server for education in modelling and simulation (MMT – Mathematics – Modelling – Tools).

Introduction - Modelling

ARGESIM Benchmark 'C7 Constrained Pendulum' is based on the dynamics of a pendulum which hits a pin: hit and release of the pin is a state event, which has to be managed properly ([1]). At *Hit* and *Release*, the pendulum changes its pivot point (Figure 1), so that the dynamics is composed of the movement of a 'long' pendulum and of a 'short' pendulum. Both movements are described by the classical nonlinear pendulum equation:

$$m \cdot l \cdot \ddot{\varphi} = -m \cdot g \cdot \sin\varphi(t) - d \cdot l \cdot \dot{\varphi}(t)$$

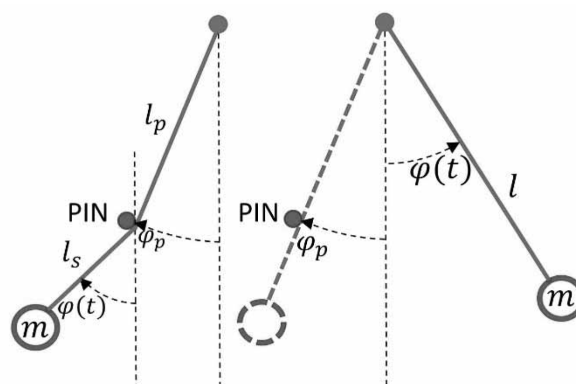

Figure 1: Sketch of the constrained pendulum.

For small angles, also the linear pendulum model is sufficient accurate. The classical linearization around the operating point $\varphi_S = 0$ is independent from angular velocity $\dot{\varphi}_S$ as the model works with linear damping:

$$m \cdot l \cdot \ddot{\varphi} = -m \cdot g \cdot \varphi - d \cdot l \cdot \dot{\varphi}(t)$$

The parameters pendulum length l, short pendulum length l_s, damping factor d, point mass m, angular pin position φ_{pin}, pin distance from pivot l_p, and initial values characterize the system.

The system is a so-called structural dynamic system ([2]), as caused by state events (*Hit* or *Release*) the dynamics change – in this case only a parameter, the pendulum length changes, and the equations remain unchanged.

The events *Hit* and *Release* obey a simple *Event Function* $e(t)$, whose zeros t_e determine the time instants of the events:

$$e(t) = \varphi(t) - \varphi_{pin} = 0 \quad \rightarrow \quad e(\varphi(t)) = \varphi - \varphi_{pin} \overset{!}{\underset{\pm}{=}} 0$$

Here the first equation is the mathematical description, the second the algorithmic: a zero search algorithm with either positive, negative, or both-sided crossing of zero.

For a dynamic system $\dot{\vec{x}}(t) = \vec{f}(\vec{x}, \vec{p}, \vec{u}, t)$ with event function $e(\vec{x}, \vec{p}, t)$, event handling generally requires the following steps within an ODE solver's integration step from t_i to t_{i+1}

- *Event Detection* by sign of event function:
  $$sign(e(t_i)) \neq sign(e(t_{i+1}))$$
- *Event Localisation* and stop of ODE solving by zero search of $e(\vec{x}(t)) = 0$ at $[t_i, t_{i+1}] \longrightarrow t_e$
- *Event Action* at t_e
- *Re-Initialisation* and re-start of ODE solving

Event actions may be simple to complex:

- *Output Event*: no event action, only time output
- *Parameter Change Event*: $p \rightarrow p^*$
- *Input Change Event*: synchronisation of input jumps with stepsize
- *State Change Event*: $x(t_e) \rightarrow x^*(t_e)$
- *Derivative Change Event*: $f(x, t) \rightarrow f^*(x, t)$
- *Model Change Event*: $\dot{\vec{x}} = \vec{f}(\vec{x}) \rightarrow \dot{\vec{z}} = \vec{g}(\vec{z})$

The constrained pendulum system with events *Hit* and *Release* involves *Parameter Change Events* and *State Change Events*. At event $e(t_e) = \varphi_{pin} - \varphi(t_e) = 0$,

- the pendulum length changes: $l \rightarrow l_s$ or $l_s \rightarrow l$,
- and due to conservation of momentum, the angular velocity changes discontinuously:
  $\dot{\varphi}(t_e) \rightarrow \dot{\varphi}\frac{l}{l_s}(t_e)$ or $\dot{\varphi}(t_e) \rightarrow \dot{\varphi}\frac{l_s}{l}(t_e)$.

Indeed it is strange, that the angular velocity, a state variable, changes discontinuously – this cannot happen in reality, it is result of simplification in modelling. This drawback can be eliminated by a simple transformation of the state, using instead of the angular velocity $\dot{\varphi}(t)$ the tangential velocity $v(t) = l \cdot \dot{\varphi}(t)$, which does not change in case of *Hit* or *Release*:

$$\dot{\varphi} = \frac{1}{l} \cdot v \qquad \dot{v} = g \cdot \sin\varphi - \frac{d}{m} \cdot v$$

1 MATLAB Model Approaches

MATLAB's ODE-solvers generally need a state space description of the model with coupled first-order differential equations, best choice for the constrained pendulum is

$$x_1 = \varphi \quad x_2 = v$$

$$\dot{x}_1 = \frac{1}{l} \cdot x_2 \quad \dot{x}_2 = -g \cdot \sin(x_1) - \frac{d}{m} \cdot x_2,$$

resulting in nonlinear state space description:

$$\dot{\vec{x}}(t) = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} \frac{1}{l} \cdot x_2 \\ -g \cdot \sin(x_1) - \frac{d}{m} \cdot x_2 \end{pmatrix} = \vec{f}(\vec{x}, t)$$

The classically linearized model – needed later - is

$$\dot{y}_1 = \frac{y_2}{l} \qquad \dot{y}_2 = -g \cdot y_1 - \frac{d}{m} \cdot y_2,$$

and reformulated as LTI state space system:

$$\dot{\vec{x}} = A \cdot \vec{x} + B \cdot \vec{u} \qquad \vec{y} = C \cdot \vec{x} + D \cdot \vec{u}$$

$$A = \begin{pmatrix} 1 & \frac{1}{l} \\ -g & -\frac{d}{m} \end{pmatrix} \quad B = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad C = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad D = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

1.1 TASK A: MATLAB Nonlinear Model with Event Handling

The first task of the benchmark is to solve the pendulum problem with an ODE-solver and to find pin touch and release with events functions.

MATLAB's ODE solvers provide event detection, but no event action handling. For events, additionally to the model derivative function $\vec{f}(\vec{x}, t)$ the event function $e(\vec{x}, t)$ can be provided.

This solution works with the classical Runge-Kutta ODE45 solver, with stepsize control. Before calling the solver, options define accuracy for step size control - 'RelTol', 1e-4, – and event specification - 'Event', @hitrelease. The solver call needs as inputs the derivative function - @pend_func - and simulation interval, initial values, and the reference to further options:

```
options=odeset('RelTol',1e-5,'Event',@hitrelease)
ode45(@pend_func, [tstart, tend], xstart, options)
```

The ODE solver can detect an event, and he can localize an event by iteration within the integration interval $[t_i, t_{i+1}] \longrightarrow t_e$ (using the *Regula Falsi* method, a combination of bisection method and secant method), resulting in a reduced integration interval $[t_i, t_e = t_{i+1}]$. There is no possibility to force *Event Actions* at event time t_e (except *Output Events*). Now the solver either re-starts the integration at $[t_e = t_{i+1}, t_{i+2}]$ and continues, or he terminates the ODE solving at t_e state with state $\vec{x}(t_e)$.

The second option, the termination at the event, is basis of the implementation for the implementation of the constrained pendulum model: a loop switches between solving the 'long' pendulum model and the 'short' pendulum model, each terminated by the *Hit* or *Release* events.

The implementation itself is quite straightforward with a while-loop, which stops if the time reaches the defined simulation end time (10 sec).

Inside the loop an *if-elseif-else* clause decides whether the long or the short pendulum system is used and appended the overall solution. The decision logic works for arbitrary initial values and pin positions, but becomes more complex for a possible special case: in case the *Hit* or *Release* event is around at pin position (within a certain numerical accuracy, the tangential velocity at event time must decide about further model selection. The following code snippet shows details of this implementation, which is a classical hybrid decomposition of the constrained pendulum model into a controlled sequence of 'long' pendulum model and 'short' pendulum model.

```
if y_start(1) > phi_p % calculating with long pendulum
   sol = ode45(dydt1, [tstart, tend], y_start, options);
   t = [t, sol.x]; y = [y, sol.y]; t_events = [t_events, sol.xe];
elseif y_start(1) < phi_p    % calculating with short pendulum
         sol = ode45(dydt2, [tstart, tend], y_start, options);
         t = [t, sol.x]; y = [y, sol.y]; t_events = [t_events, sol.xe];
else
if y_start(1) > phi_p % calculating with long pendulum
   sol = ode45(dydt1, [tstart, tend], y_start, options);
   t = [t, sol.x]; y = [y, sol.y]; t_events = [t_events, sol.xe];
elseif y_start(1) < phi_p    % calculating with short pendulum
      sol = ode45(dydt2, [tstart, tend], y_start, options);
     t = [t, sol.x]; y = [y, sol.y]; t_events = [t_events, sol.xe];
else
if y_start(2) < 0    % calculate with short pendulum
   sol = ode45(dydt2, [tstart, tend], y_start, options);
   t = [t, sol.x]; y = [y, sol.y]; t_events = [t_events, sol.xe];
else    % calculate with long pendulum
   sol = ode45(dydt1, [tstart, tend], y_start, options);
   t = [t, sol.x]; y = [y, sol.y]; t_events = [t_events, sol.xe];
end; end
```

The model derivative functions can be defined as inline function by

```
dydt1 = @(t,y)[y(2)/l; -g*sin(y(1))-d/m*y(2)];
dydt2 = @(t,y)[y(2)/ls; -g*sin(y(1))-d/m*y(2)];
```

The algorithmic event function has as parameters the event function 'value' itself, the stopping flag 'is terminal=1' to stop ODE solving at the event, and 'direction=0' to detect *Hit* and *Release*:

```
function [value,isterminal,direction] = hitrelease(~,y)
   value = y(1)-phi_p;
   isterminal = 1; direction = 0;
end
```

Figure 2 shows the results for the 'standard' initial values $\varphi_0 = \pi/6,\ \dot{\varphi}_0 = 0, \varphi_{pin} = -\pi/12$. Event times are:

0.7035 1.1518 2.5904 2.9905 4.5427 4.8675 6.6487 6.7204

Obviously the fourth contact (7th event)results in a very short window for the 'short' pendulum, and may cause 'event vanished' for too big stepsizes.

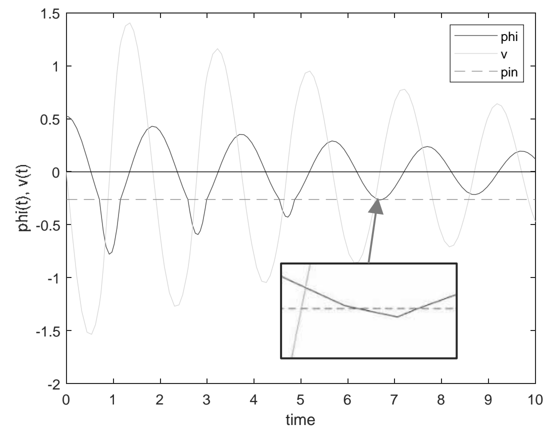

Figure 2: ODE45 solutions for $\varphi(t)$ and $v(t)$ for 'standard' initial values with detail for last two events rel. tolerance 1e-4, max. stepsize 0.15.

Important for the accuracy of event finding is the stepsize control of the ODE solver. ODE45 estimates the local error by the difference of a 4th order step and a 5th from t_i to $t_{i+1} = t_i + h$: exceeding the given relative tolerance, the stepsize decreases to h^-, $t_{i+1} = t_i + h^-$, a too big undercut increases the stepsize to h^+, $t_{i+1} = t_i + h^+$.

After the choice of a proper stepsize the event finding starts - with an accuracy depending on ODE solver accuracy and general accuracy eps. A small stepsize brackets a small interval for fast event finding, but may result in slow ODE solving. A too big stepsize may cause problems: events may vanish, as in this case with the forth pin contact: here the bracketed interval for event finding may be too large, so that both events are within the window and will therefore not be detected.

1.2 TASK B: MATLAB Linear Model – ODE Solver with Event Handling

Task is to compare the nonlinear model with the linear model. For the linear model also the event finding features of the ODE solver can be used, so that the implementation simply replaces the nonlinear model from Task A with the linear one:

For graphical comparison, both linear and nonlinear solutions are plotted into one graphic window. Figure 3 displays both results for the 'standard' initial values, showing only slight differences in the event times. Event times are summarizes in Table 3, Section 5, for better comparison.

Of course the implementation works also for the 'original' smaller initial values foreseen for this task, $\varphi_0 = \pi/12,\ \dot{\varphi}_0 = 0, \varphi_{pin} = -\pi/24$, resulting in even smaller differences of nonlinear/linear event times.

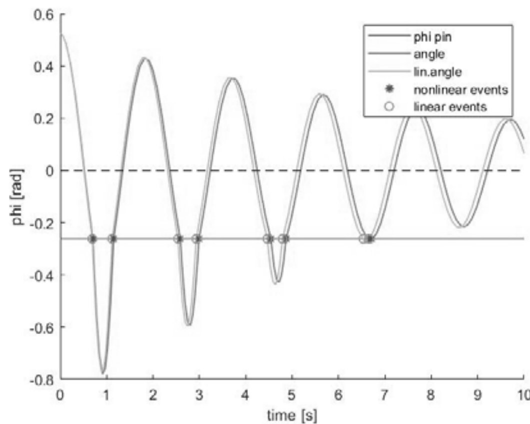

Figure 3: ODE45 solution for linear and nonlinear system in MATLAB with event finding (rel.tol 1e-5).

It is to be noted, that simulation of nonlinear and linear system results in different time bases, because of differences in the step size control. For a numerical comparison, e.g. difference of the angles φ(t) and $\varphi_{lin}(t)$, the time bases must be interpolated after the simulation. One could force the ODE solvers to a given (output) time base, but then problems with the event times occur.

For a real precise comparison of the full time courses, both models must run in parallel with state vector $(\varphi, v, \varphi_L, v_L)^T = (x_1, x_2, x_3, x_4)^T$, with an extended event control of the linear and of the nonlinear system using a vector event function:

$$\vec{e}\big(\varphi(t), \varphi_L(t)\big) = (\ \varphi(t) - \varphi_{pin} \quad \varphi_L(t) - \varphi_{pin})^T$$

The model becomes a joint model implemented as

```
dydt1 = @(t,y)[y(2)/lnlakt; -g*sin(y(1))-d/m*y(2)];
dydt2 = @(t,y)[y(2)/lnlakt; -g*sin(y(1))-d/m*y(2)];
dydt3 = @(t,y)[y(4)/llakt; -g*y(3)-d/m*y(4)];
dydt2 = @(t,y)[y(2)/llakt; -g*y(3)-d/m*y(4)];
```

Now the loop, switching, and concatenating of the sequence of models becomes more complex: in each event the next actual length can be any combination, as events are linear long – linear short & nonlinear long – nonlinear long, ... The algorithmic event function must work with two event entries:

```
function [value,isterminal,direction] = hitrelease(~,x)
value(1) = x(1)-phi_p;  value(2) = x(3)-phi_p
 isterminal(1) = 1; direction(1) = 0;
 isterminal(2) = 1; direction(2) = 0;
end
```

This procedure seems complicated, but it is the general event handling strategy used in Simulink, and therefore useful to study.

1.3 MATLAB Nonlinear Model without Event Handling

The loop, switching, and concatenation of ‘long’ pendulum and ‘short’ pendulum is indeed laborious – why not to change the length directly in the algorithmic pendulum function, depending on angle position ?

This quick and ‘dirty’ approach has ‘strange’ results. The model function for both models, using MATLAB’s effective abbreviations for *if-then-else* clauses, becomes

```
function dxdt = pend_noev_fun(~,x)
  lakt = (x(1) >= phi_p)*l +(x(1) <= phi_p)*ls
  dxdt(1) = x(2)/lakt;
  dxdt(2) = -g*sin(x(1))-d/m*x(2); end;
sol = ode45(@pend_noev_func, [tstart, tend], xstart, options),
```

and the simulation call consist only of one call of the ODE solver. The results are astonishing close to the simulation with events handling, shown in Table 1 (event times for the ‘standard’ initial values), with unexpected results.

Phase Start	Event Times		
	Event Finder	No Event Finder	
	rtol 1e-4	rtol 1e-5	rtol 1e-4
Long 1	0.0	0.0	0.0
Short 1	0.703459556	0.703459559	0.702954406
Long 2	1.151778788	1.151778616	1.157402743
Short 2	2.590418102	2.590358975	2.583773000
Long 3	2.990527098	2.990509554	2.998855259
Short 3	4.542743634	4.542667578	4.535188672
Long 4	4.867485452	4.867455379	4.874065441
Short 4	6.648742768	6.648572636	
Long 5	6.720351405	6.7204086952	

Table 1: Event times with and without event detection – with vanishing events and unexpected event sequence.

What results are to be expected? Generally, without event finder, the ODE solver recognizes the necessary change of the length at the next integration time instant, i.e. at t_{i+1} definitively too late - it should have happened at unknown $t_e, t_i < t_e < t_{i+1}$

Figure 4 explains the situation, showing both solutions around an event time, taking into account the different stepsizes of ODE solver with event handling t_i^e, t_{k+1}^e and without event handling $t_{k-1}^n, t_k^n, t_{k+1}^n$. Obviously the solver without event finding chooses for the same given tolerances shorter stepsizes around the event.

The reason is a numerical problem: the jump of the length makes the ODE function discontinuous, and the ODE solver tries to keep the tolerances, decreasing the stepsize – in vain: he ends up with $t_{k+1}^n = t_e^n$, violating the tolerances (hidden warnings).

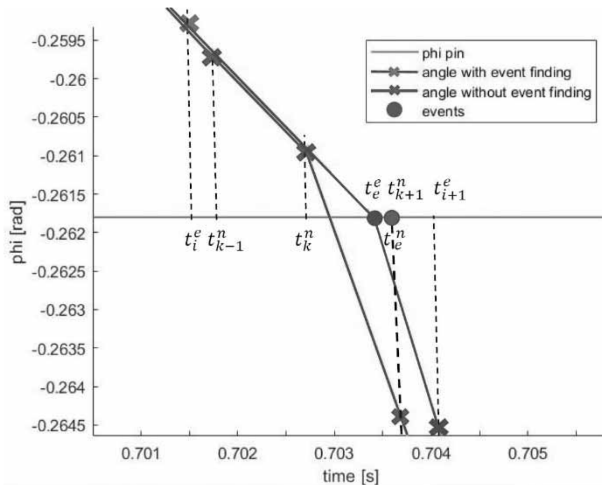

Figure 4: Operation of ODE solver with and without event detection, expected event sequence.

But interestingly the results seem plausible, because the comparisons of time instants

$$t_i^e < t_{k-1}^n < t_k^n < t_e^e < t_{k+1}^n = t_e^n < t_{i+1}^e$$

shows the expected behaviour, 'correct' event before 'faked' event: $t_e^e < t_{k+1}^n = t_e^n$.

Table 1 – event times (t_e^e) with and without (t_e^n) event detection and different ODE tolerances – shows expected numerical values, but only for some event times (denoted in green). Some other event times t_e^n without event detection take place before the correct event ($t_e^n < t_e^e$, denoted in red). This unexpected result is caused by the 'failing' stepsize control, which for higher tolerances takes 'too small' stepsizes, so that the 'correct' event lies after the 'faked' event (Figure 5):

$$t_i^e < t_{k-1}^n < t_k^n < t_{k+1}^n = t_e^n < t_e^e < t_{k+2}^n < t_{i+1}^e$$

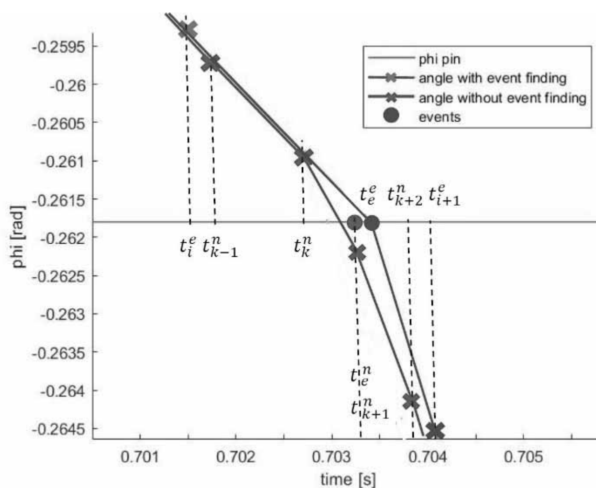

Figure 5: Operation of ODE solver with and without event detection, unexpected event sequence.

Which event time is now the correct one – t_e^e, or t_e^n? Indeed the 'exact' event time t_e^e is not exact, it is a numerical approximation. Curiously the quick and 'dirty' implementation with the discontinously changing length can give a better result $t_e^n < t_e^e$, misapplying the failing stepsize control as 'pseudo-event-finder'. But Table 1 shows for low tolerances definitely wrong results, with vanishing events for this strategy. But this strategy must be used, if no event detection is available (as in case of EXCEL, [3]), but only with extreme care. As consequence, event finding is necessary, but it has to be 'synchronised' carefully with tolerance parameters of the ODE solver.

1.4 MATLAB Linear Model with LTI Solving

The linear model is appropriate for small angles, and for time analysis an ODE solver is not the best approach (only approximating the time course). The linear pendulum is an LTI system, and therefore the linear theory with the exponential matrix provides a powerful tool, which is exact with respect to the algorithmic error:

The classically linearized model with reformulation as LTI state space system is

$$\dot{y}_1 = \frac{y_2}{l} \qquad \dot{y}_2 = -g \cdot y_1 - \frac{d}{m} \cdot y_2,$$

$$\dot{\vec{x}} = A \cdot \vec{x} + B \cdot \vec{u}, \quad A = \begin{pmatrix} 1 & \frac{1}{l} \\ -g & -\frac{d}{m} \end{pmatrix} \quad B = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Linear theory derives a solution using the exponential matrix

$$\dot{\vec{x}}(t) = e^{A \cdot t} \cdot \vec{x}_0 + \int_0^t e^{A \cdot (t-\tau)} \cdot B \cdot u(\tau) d\tau$$

The properties of the exponential matrix allow to calculate a solution recursively on a time grid by

$$\vec{x}(t_{k+1}) = \vec{x}_{k+1} = e^{A \cdot h} \cdot \vec{x}_k, \qquad h = t_{k+1} - t_k$$

MATLAB offers with the LSIM solver an integrated tool for solving LTI systems by

```
sol=lsim( A, B, C ,D, timegrid )
```

using the state update with the exponential matrix, but without event finding capabilities. So it is now the task to combine the linear exact method with event finding.

Trying to use LSIM, one possibility would be to simulate only one single time step per iteration. After each one-timestep simulation with LSIM, the resulting angle gets checked for crossing the pin angle, before it gets

written into a consistent result vector or the event gets estimated with one Newton-algorithm step.

However, this method is very inefficient. Each call of the LSIM forces a new calculation of the exponential matrix for every time step.

Another possibility is to simulate with LSIM longer time periods in a while-loop, and run through the solution vector to check for the event. If the angle crosses the pin angle within the solution vector a Newton-algorithm step gives the estimated event time and only the part of the solution vector until the event gets used – and the while loop continues. This method however can easily get a bit confusing or chunky to implement.

The best method – and presented here - is indeed to calculate only one timestep and check for the event per loop iteration, but not by means of LSIM, but by direct use of the recursion with the exponential matrix - this makes a clearer implementation and reduces unnecessary evaluations and recalculations of the exponential matrix. The event finding is a heuristic Newton implementation: it performs only one iteration, but with exact derivative calculation: $\dot{\varphi}(t^*) = v(t^*)/l$ is generically given by the ODE.

The implementation with while-loop and decision logic for choice of the next pendulum length is similar to the ODE approach with event finder; additionally the event finding is done by the Newton heuristics:

```
% calculate the exponential matrices
    A_expm = expm(A*tstep);
    A_red_expm = expm(A_red*tstep);

  while sol.t(end) < tend          % rewrite initial conditions
  y_start = sol.y(:,end);t_start = sol.t(end);
    if y_start(1) < phi_p    % calculate with short pendulum
    [t,y_new] = expsolve(y_start, t_start, A_red_expm); l_ind = ls;
  elseif y_start(1) > phi_p     % calculate with long pendulum
    [t,y_new] = expsolve(y_start, t_start, A_expm); l_ind = l;
  else    % consider velocity direction
    if y_start(2) < 0     % calculate with short pendulum
    [t,y_new] = expsolve(y_start, t_start, A_red_expm); l_ind = ls;
  else     % calculate with long pendulum
    [t,y_new] = expsolve(y_start, t_start, A_expm); l_ind = l;
end; end

% detect event and perform Newton approximation
  if (sol.y(1,end)-phi_p)*(y_new(1)-phi_p) < 0
    t_event = sol.t(end) + (phi_p - sol.y(1,end))/…
                (sol.y(2,end)/l_ind);
    y_event = [phi_p; (y_new(2)-sol.y(2,end))/(y_new(1)-
                sol.y(1,end))*(phi_p-sol.y(1,end)) + sol.y(2,end)];
    sol.y = [sol.y, y_event]; sol.t = [sol.t, t_event];
    event_times = [event_times, t_event];
  else       % no event happening
    sol.y = [sol.y, y_new]; sol.t = [sol.t, t];
end; end

  function [t,y] = expsolve(y_start, t_start, exp_matrix)
    y = [exp_matrix*y_start];
    t = t_start+tstep; end
```

The results for the 'standard' initial values are very close to those of the results from the ODE solution of the linear model, time events are:

0.6920 1.1205 2.5409 2.9318 4.4659 4.7909 6.5325 6.6528

It is to be noted, that the LTI approach with the exponential matrix is a numerical exact method – so these event times may be more reliable as results with the ODE solver. Also the exponential matrix can be computed numerical exact via eigenvalues, etc. MATLAB operates with a very sophisticated environment for calculation of the experimental matrix – see 'Nineteen dubious ways to compute the matrix exponential' – [4].

1.5 MATLAB Analytical Model Approach

The linear model is indeed appropriate for small angles, and for time analysis the LTI algorithm with the exponential matrix is very useful in applications. But the pendulum system is a small one, so another approach can make use of the analytical solution, in combination with an appropriate event finding algorithm. This task requires symbolical and numerical computations, and the following investigations deal with three approaches.

Analytic-Numeric Approach

This approach makes directly use of the known analytical solution, a closed formula to be evaluated at arbitrary time instants:

$$\varphi_{sym}(t) = \varphi_{sym}(t, l, \varphi_0, v_0) =$$
$$= e^{-\omega_0 * D * t}[c_1 \cos(\omega_0 \cdot \delta \cdot t) + c_2 \sin(\omega_0 . \delta \cdot t)]$$
$$c_1 = \varphi_0, \omega_0 = \sqrt{\frac{g}{l}}, D = \frac{d \cdot \omega_0}{2m}, \delta = \sqrt{1 - D^2}, c_2 = \frac{v_0}{l \cdot \omega_0 \cdot \delta}$$

and with related tangential velocity $v_{sym}(t, l, \varphi_0, v_0)$.

The analytical (symbolic) solution depends on the pendulum length, and on the initial values, which change in case of event *Hit* or *Release*: $\varphi_{sym}(t, l_e, \varphi_{t_e}, v_{t_e})$.

Again the event function $e(t) = \varphi(t) - \varphi_{pin}$ is used, but now inserting the analytical symbolic solution valid since the previous event $t_{e,p}$ with actual length $l_{e,p}$ chosen at previous event:

$$e(t) = \varphi_{sym}\left(t, l_{e,p}, \varphi_{t_{e,p}}, v_{t_{e,p}}\right) - \varphi_{pin}$$

Starting now with an appropriate guess $t_{e,n}^{[0]}$ for the next event time, a Newton iteration recursively tries to determine the zero of the event function:

$$t_{e,n}^{[k+1]} = t_{e,n}^{[k]} - \frac{e\left(t_{e,n}^{[k]}\right)}{\dot{e}\left(t_{e,n}^{[k]}\right)} =$$
$$= \frac{l_{e,p} \cdot \left(\varphi_{sym}\left(t_{e,n}^{[k]}, l_{e,p}, \varphi_{t_{e,p}}, v_{t_{e,p}}\right) - \varphi_{pin}\right)}{v\left(t_{e,n}^{[k]}, l_{e,p}, \varphi_{t_{e,p}}, v_{t_{e,p}}\right)}$$

Again the necessary derivative is generically given by the tangential velocity, and the resulting MATLAB implementation is simpler than the iteration formula. Again a while-loop performs the iteration, and interestingly four iterations are sufficient to result in event times as accurate as calculated by other methods:

```
while iterations<4
  newton_time=before-((part_sol_phi-phi_p)/part_sol_v);
  part_sol_phi = exp(-alpha_red*newton_time)*…
        ((C1*cos(w_red*newton_time))+…
        (C2_red*sin(w_red*newton_time)));
  part_sol_v = exp(-alpha_red*newton_time)*…
         ((((w_red*C2_red)-(alpha_red*C1))*…
         cos(w_red*newton_time))-…
        (sin(w_red*newton_time)*((w_red*C1)+…
        (alpha_red*C2_red))));
  before=newton_time;
  iterations=iterations+1;
end
```

The iteration loop runs in a while-loop switching between 'long' and 'short' pendulum: a simple binary counter decides which pendulum length is to be used.

Analytic-Symbolic Approach

This approach again makes directly use of the known analytical solution, a closed formula to be evaluated at arbitrary time instants:

$$\varphi_{sym}(t) = \varphi_{sym}(t, l, \varphi_0, v_0), \ \ v_{sym}(t) = v_{sym}(t, l, \varphi_0, v_0)$$

Task is to determine the events, i.e. the zeros of the event function by means of the event function valid since the previous event $t_{e,p}$ with actual length $l_{e,p}$, chosen at previous event:

$$e(t) = \varphi_{sym}\left(t, l_{e,p}, \varphi_{t_{e,p}}, v_{t_{e,p}}\right) - \varphi_{pin}$$

But now the symbolic solution is inserted directly, so that a nonlinear equation for the next event time $t_{e,n}$ arises:

$$e(t_{e,n}) = \varphi_{sym}\left(t_{e,n}, l_{e,p}, \varphi_{t_{e,p}}, v_{t_{e,p}}\right) - \varphi_{pin} = 0$$

$$e^{-\omega_0 * D * t}\left[c_1 \cos(\omega_0 \cdot \delta \cdot t_{e,n}) + \right.$$
$$\left. + c_2 \sin(\omega_0 \cdot \delta \cdot t_{e,n})\right] - \varphi_{pin} = 0$$

$$c_1 = \varphi_0, \omega_0 = \sqrt{\frac{g}{l}}, D = \frac{d \cdot \omega_0}{2m}, \delta = \sqrt{1 - D^2}, c_2 = \frac{v_0}{l \cdot \omega_0 \cdot \delta}$$

It is laborious to solve this equation with respect to $t_{e,n}$ 'manually', but MATLAB provides with the *Symbolic Toolbox* an adequate tool. Defining the event time $t_{e,n}$ as symbolic variable, and the error function as symbolic equation, MATLAB's vpasolve tool indeed masters this task. After solution, the symbolic event time can be numerically evaluated. The implementation is quite short, and results in almost equivalent results with other approaches – see Table 3, Section 5.

```
syms t  equa
C1=phi0;  C2=(v0+(alpha*phi0*l))/(w*l);  %constants
equa=exp(-alpha*t)*((C1*cos(w*t))…
        +(C2*sin(w*t)))==phi_p;     %equation for phi=phi_p
te_sym = vpasolve( equa , t);  te = double(te_sym)
```

Full Symbolic Approach

For this approach the *Symbolic Toolbox* also sets up the analytical solutions $\varphi_{sym}(t)$ and $v_{sym}(t)$ by solving the ODEs analytically. Therefore, the state variables must be implemented as symbolic functions, as well as the differential equations. The following implementation documents the symbolic automatized operations:

```
syms phi(t) v(t)   %work with symbolic variables
%differential equations with symbolic values
  eqns = [diff(phi,t) == v/l, diff(v,t) == -g*phi-d*v/m];
  eqns_red=[diff(phi,t) == v/ls, diff(v,t) == -g*phi-d*v/m];
  cond=[phi(0)==solution_phi(end),v(0)==solution_v(end)];
%solve differential equations
  if n==0   structure=dsolve(eqns,cond); n=1;
  else structure=dsolve(eqns_red,cond);  n=0; end
```

The ODEs are solved with the dsolve tool which returns a symbolic time-dependent solution. As in the *Analytic-Symbolic Approach*, the symbolic vpasolve tool allows to determine the event times, using now the symbolic ODE solutions, and not the 'manually' derived solutions – very comfortable. With a *for-loop* the symbolic ODE solutions are evaluated till the event time with a defined time step and transformed to a numerical value.

But because every evaluation step includes a transformation from symbolic to numeric value, this approach takes much longer time in MATLAB.

Comparison of Analytic Approaches

All analytic approaches provide the 'exact' same results for the event times (within rounding tolerances):

0.6920 1.1205 2.5409 2.9318 4.4658 4.7908 6.5321 6.6530

But the calculation time duration differs significantly:

- the *Analytic-Numeric Approach* has the shortest calculation time (0.1-0.25 seconds),
- the *Analytic-Symbolic Approach* takes 10 times longer (2-5 seconds),
- and the *Full Symbolic Approach* has the longest time (15-17 seconds).

For the used 'standard' initial values $\varphi_0 = \pi/6$, $\varphi_{pin} = -\pi/12$, the event times are close to the event times of the nonlinear system. For smaller initial values $\varphi_0 = \pi/12$, $\varphi_{pin} = -\pi/24$ they get very close (Task B).

An interesting alternative for getting generally closer to the nonlinear behaviour would be the use linear affine systems with different linearization points.

1.6 TASK C: Boundary Value Problem

The benchmark defines as third task the solution of a boundary value problem: which initial angular velocity $\dot{\varphi}_0$ with initial angle $\varphi_0 = \pi/6$ and pin angle $\varphi_{pin} = -\pi/12$ is necessary to reach after one *Hit* event exactly $\varphi_{target} = -\pi/2$ (includes $\dot{\varphi}_{target} = 0$) ?

The classic approach is an iteration of $\dot{\varphi}_0$ or $\dot{v}_0$, resp., each with a simulation run with 'sufficient' time length. A charming easier alternative is to view the problem from the target values: starting with the (short) pendulum at angle $\varphi_{target} = -\pi/2$ and velocity $\dot{\varphi}_{target} = 0$ backwards in time, the pendulum will reach after one *Release* event as long pendulum at a certain time $t_e^{[\pi/6]}$ the angle $\varphi\left(t_e^{[\pi/6]}\right) = \pi/6$ with wanted velocity $\dot{\varphi}(t_e^{[\pi/6]})$.

The dynamics backwards in time can be derived by a time transformation $\tau = -t$ yielding $d\tau = -dt$ and

$$\frac{d}{d\tau}\varphi(\tau) = -\frac{1}{l}\cdot v(\tau), \frac{d}{d\tau}v(\tau) = -g\cdot\sin\varphi(\tau) + \frac{d}{m}\cdot v(\tau)$$

As consequence, the boundary value problem changes to a simulation backwards in time, with *Release* event and with a *Target* event at $t_e^{[\pi/6]}$. The event function becomes a vector event function:

$$\vec{e}(\varphi(t)) = (\varphi(t) - \varphi_{pin} \quad \varphi(t) - \pi/6)^T$$

The implementation for the nonlinear model with event handling can be re-used, changing only the signs in the model function and adding the *Target* event:

```
function [value,isterminal,direction] = hitrelease(~,y)
 value  = [y(1)-phi_p    y(1)-pi/6];
 isterminal = [1  0]; direction = [0 -1]; end
```

Note, that the *Target* event does not terminate the simulation, it stores only the state target values (Figure 6, events *Release* (green cross) and *Target* (red cross).

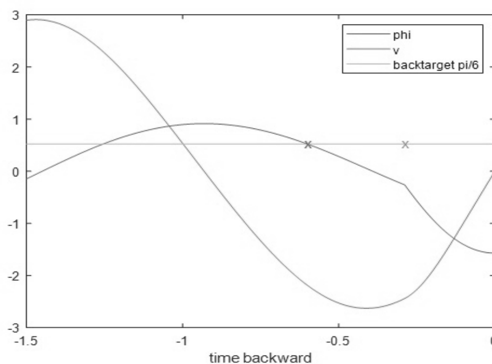

Figure 6: Simulation backwards in time for determining wanted angular velocity $\dot{\varphi}_0$.

The numerical results are quite accurate:

$$t_e^{[\pi/6]} = 0.601299, \dot{\varphi}(t_e^{[\pi/6]}) = v\left(t_e^{[\pi/6]}\right) = -2.184514$$

2 Simulink Model with and without Event Handling

Simulink is MATLAB's graphical modelling environment based on directed signal flow and classic block-oriented modelling. Basis is the *Integrator Block* – denoted by $\int or \frac{1}{s}$, which 'integrates' the input, the ODE. In principle, Simulink makes use of the integral notation of ODEs, in case of the pendulum of

$$\varphi = \int\left(\frac{1}{l}\cdot v\right) \quad v = \int\left(-g\cdot sin\varphi - \frac{d}{m}\cdot v\right)$$

The Simulink model of the constrained pendulum is based on the pendulum ODEs using two integrator blocks.

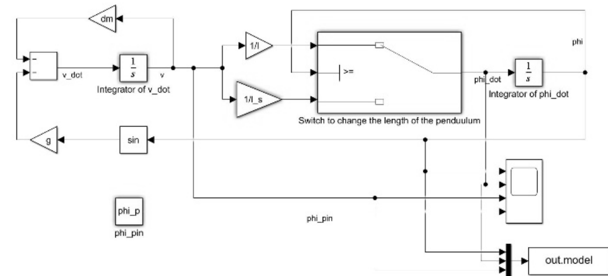

Figure 7: Simulink model for the constrained pendulum with switch block for changing length.

Due to continuous differentiability of the states φ and v, only the length of the pendulum must be changed, depending on the angle φ. There, Simulink offers a S*witch* block, which compares whether the current angle φ is greater than pin angle φ_p (the threshold is defined in the switch block), and which therefore models the event function (Figure 7).

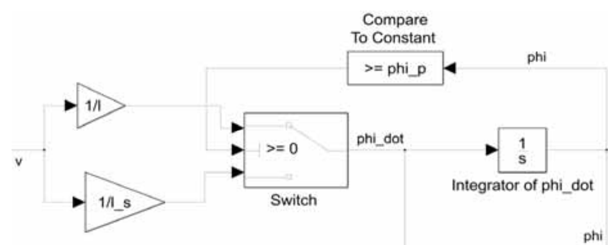

Figure 8: Simulink model for the constrained pendulum with compare block and switch block for changing length; detail.

A more detailed approach for the event function is the use of a *Compare to Constant* block in addition to a switch block. Since the output of this block is Boolean, the threshold of the consecutive switch is set to 0 (Figure 8). However, both switching options deliver the same results.

Simulink offers state event handling by means of the *Zero Crossing* options in many blocks (23 blocks !), as e.g. in the *Hit Crossing* block (for general event functions), and in *Compare* and *Switch* blocks (for simple event functions).

It is possible to activate and deactivate the zero-crossing detection (Figure 9, switch block), so simulation can run with and without event detection.

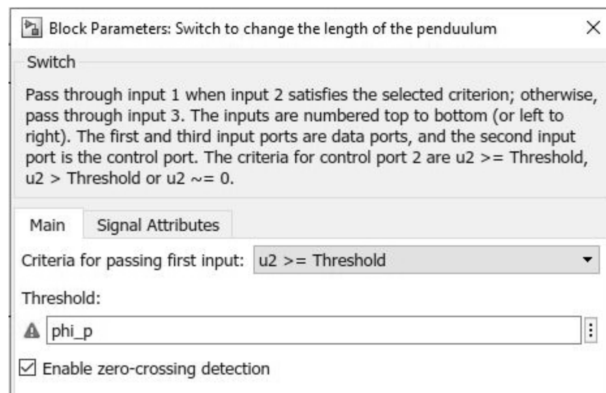

Figure 9: Configuration menu for switch block with threshold definition and zero crossing enabling and disabling.

Simulink parses the graphic model and compiles it into state space model $\dot{\vec{x}}(t) = \vec{f}(\vec{x}, t)$. For simulation it makes use of the MATLAB ODE solver suite, quite similar to the use in MATLAB, but with extended possibilities for events, triggered or enabled/disabled submodels, and some other special tasks.

Indeed Simulink does state event handling, and offers – in contrary to MATLAB – also possibilities to implement *Event Actions*. Generally, for this purpose there are two possibilities:

- *Parameter Change Events* and *State Change Events* can be directly described in one model by switches and re-initialisation of integrator blocks, triggered by blocks capable of zero crossing detection.
- *Derivative Change Events* and *Model Change Events* need another technique: original and changed derivatives or original and changed models resp., are both put into different Simulink submodels, which can be enabled or disabled. The 'root'model handles via events the switching between the submodels by enabling or disabling them.

The above submodel approach for structural-dynamic systems is the so-called *Monolithic State Space Approach* ([2]), the alternative to the *Hybrid Decomposition Approach*, used also in MATLAB (see Section 1.1).

The term monolithic refers to the fact that the state space is a maximal one and not consecutively split into smaller state spaces: during simulation, in disabled submodels the states are 'frozen', and re-activated, as soon as the submodel is enabled.

In this Simulink model for the constrained pendulum the 'one model' approach is chosen, a generic simple approach with state vector $(\varphi(t)\ \ v(t))^T$ and switching length. The more general alternative would work with a monolithic overall state

$$\left(\varphi_l(t)\ \ v_l(t)\ \ \varphi_{l_s}(t)\ \ v_{l_s}(t)\right)^T$$

– used in the Stateflow modelling approach, Section 3.

State Event Detection – Zero Crossing

Simulink allows to enable or disable event handling. Simulink's event detection is more sophisticates than the MATLAB algorithm. Again for event detection the *Regula Falsi* method, a combination of bisection method and secant method, is used, with a 'hard' accuracy limit – in case of the *non-adaptive* strategy; recent Simulink versions offer an *adaptive* strategy, which instead of the 'hard' limit works with an appropriate threshold around zero, stopping the detection algorithm.

The investigations here refer to enabled (non-adaptive) zero crossing and disabled zero crossing (no event detection), and show as with MATLAB astonishing but different results. Key parameters for localisation and accuracy are again the parameters for the ODE45 solver, the tolerances and the maximal stepsize.

Generally, the results are very close to the MATLAB results – time courses as well as event times – with and without event handling.

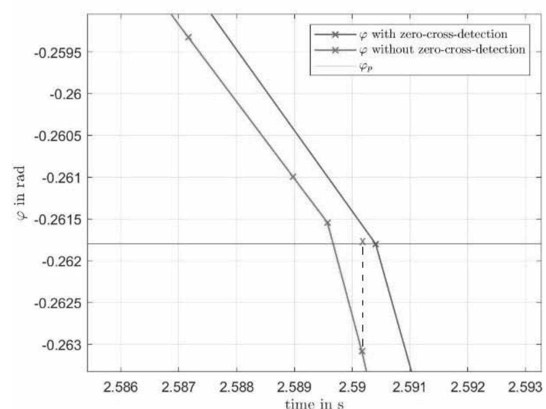

Figure 10: Solution with and without zero crossing near event: blue cross – detected event time, red cross – 'accepted' late event time.

There are small, but crucial influences of the algorithmic parameters, yielding astonishing results. Figure 10 displays in detail the ODE45 calculations around an event time, with enabled/disabled zero crossing:

- Without event detection: ODE45 varies the stepsize, and close to reaching the angle φ_{pin},
  the solver reduces the step size because the state forecast with switched length exceeds the given tolerance;
  after some tries (more solution points before event), the solver ends up with an internal error warning and accepts the 'next' time instant as event time $t_e^n = t_{n+1}$ (theoretically after the event).
- With event detection: the ODE45 solver approaches the event with bigger stepsizes. After detection of the event, event localisation starts and results in 'exact' event time t_e^e (theoretically before the time instant calculated without zero crossing algorithm: $t_e^e < t_e^n$).
- Comparison: with fixed stepsizes, the event time sequence must obey $t_e^e < t_e^n$;
  with step size control, without zero crossing algorithm, the stepsize control decreases the stepsizes, so that the 'accepted' event time $t_e^n = t_{n+1}$ may be before (!) the exact time: $t_e^n < t_e^e$;
  for further details, see discussion in Section 1.4).

Indeed the stepsize control based on tolerances and stepsize limits yields this astonishing results. From another viewpoint, stepsize control could be seen as competitive event handling, searching for a stepsize which tolerance reaching event. Some observations:

- Using the ODE45 solver with maximal stepsize set to automatic, only three hits of the pin are found. The amount of touches found depends on the chosen maximum stepsize and the tolerances.
- For the ODE45 with enabled zero-cross-detection with a maximum step size smaller than 0.154 and a relative tolerance of 10^{-4} the pendulum hits the pin four times (Figure 11).
- In contrast to this, for the ODE45 without zero-cross-detection, since it is not as accurate, the maximum step size needs to be smaller than 0.145 to observe four hits.

Further results, especially a comparison between the different approaches and resulting ecent times, see Section 5, Table 3.

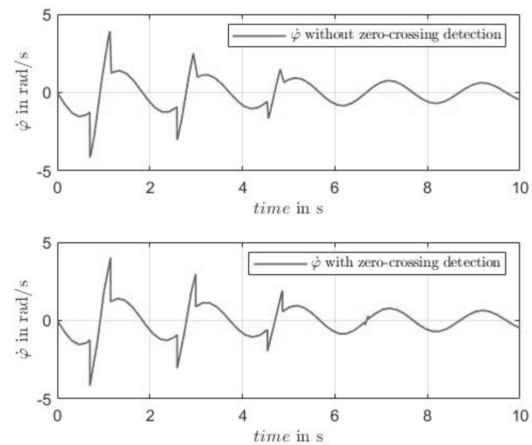

Figure 11: Angular velocity with disabled and enabled zero crossing, solver option maximum stepsize set to 0.154, rel. tolerance 10^{-4}; disabled zero crossing lets 7th and 8th event vanish.

3 Simulink Stateflow Model with and without Event Handling

Stateflow is a Simulink extension offering control schemes of signals and submodels by automata. Recent versions of Stateflow allow not only logic states in the automata, but also hybrid continuous states.

Use of Stateflow for the constrained pendulum model could on the one side simply replace the switch block of the Simulink implementation in Section 2 by one state chart 'actual' length' alternatively switched via feedback with switch of length. On the 'advanced' side, Stateflow allows to implement the constrained pendulum system as structural-dynamic system by the monolithic state space approach ([2]) using indeed the maximal state space

$$\left(\varphi_l(t)\ \ v_l(t)\ \ \varphi_{l_s}(t)\ \ v_{l_s}(t)\right)^T$$

The implementation is based on two almost identical Simulink submodels (Figure 13, a) and b)) with the pendulum system. Stateflow (Figure 12) switches between these two models, one with length l and one with length l_s.

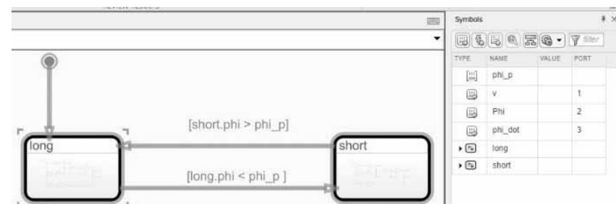

Figure 12: Overall Stateflow model with two hybrid states 'long' and 'short'.

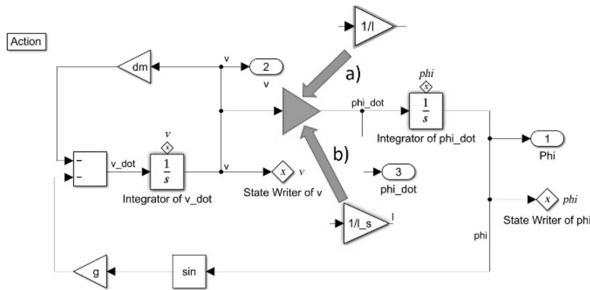

Figure 13: Model of the long pendulum (a) in the hybrid state 'long', and of short pendulum (b) in hybrid state 'short'. StateWriter blocks transfer the system states φ and v between the hybrid states.

The Stateflow model (Figure 12) includes the hybrid states 'long' and 'short'. The arrows in between mark the switch between the hybrid states: the conditions, when to switch (event function zero crossing) are given above the arrows in the squared brackets. The two hybrid states consist of Simulink submodels with the continuous dynamics (Figure 13, a) and b)).

The transition condition is given by the change of sign in the event function. The transition condition from the long pendulum to the short one is $\varphi_{long} < \varphi_{pin}$. As soon as the inequation is satisfied, the computation is done in the model of the short pendulum, and the system states of the long pendulum are frozen. To switch from the short pendulum to the long pendulum model, the inequation $\varphi_{short} > \varphi_{pin}$ needs to be fulfilled.

To start with the correct initial values after changing the model, a *StateWriter* block is used transferring the values at event time. Although the default model is the long pendulum (defined by the root arrow to state 'long', the simulation still works if the initial angle $\varphi_0 < \varphi_{pin}$. This is based by the order of work steps in Simulink Stateflow: if a new hybrid state, in this case a dynamic model, is entered, Simulink checks if a transition condition is fulfilled, in which case the transition is done before the calculation in this hybrid state starts.

An important tool in Stateflow is the *Symbols Panel*, (Figure 12, at right). In this panel the variables used in the Stateflow model can be defined as 'constant data', 'parameter data', 'local data', etc. In case of the constrained pendulum 'phi_p' is defined as a 'parameter data' and 'v', 'phi' and 'phi_dot' are defined as 'output data'.

As the Stateflow implementation clearly makes use of the same zero crossing as in the Simulink implementation, the results – time courses and event times – are very close. Further results, especially the comparison between the different approaches can be found in Table 3, Section 5.

4 System Sensitivity

Time domain analysis is the primary tool for analysis of dynamic systems $\dot{\vec{x}}(t) = f(\vec{x}, t)$. But as systems depend also on parameters $\vec{p}$, so systems and solutions are also functions of the parameters: $\dot{\vec{x}}(t, \vec{p}) = f(\vec{x}, \vec{p}, t), x(t, \vec{p})$.

Sensitivity Analysis is a method, which qualifies and quantifies the change of the solutions (or key measures of the solution) with respect to change of the parameters – and the two method groups are analytical methods and stochastic methods.

4.1 Parameter Sensitivity Functions

The so-called *Parameter Sensitivity Functions* with the *Parameter Sensitivity Equations* are a classical tool for analysing the dynamics of an ODE system $\dot{\vec{x}}(t) = f(\vec{x}, \vec{p}, t)$ with respect to change of parameters $\vec{p}$.

The sensitivity functions are generally the partial derivatives of the states $x_i(t)$ with respect to the parameters p_k, obeying the sensitivity equations, ODEs coupled with the system equations and derived by valid change of the derivation sequence:

$$x_{i,p_k}(t) = \frac{\partial}{\partial p_k} x_i(t) \quad \dot{x}_{i,p_k}(t) = \frac{\partial}{t} \frac{\partial}{\partial p_k} x_i(t)$$

$$\frac{\partial}{t} \frac{\partial}{\partial p_k} x_i(t) = \frac{\partial}{\partial p_k} \frac{\partial}{t} x_i(t) = \frac{\partial}{\partial p_k} \dot{x}_i(t) = \frac{\partial}{\partial p_k} f_i(\vec{x}, t)$$

$$\dot{x}_{i,p_k}(t) = \frac{\partial}{\partial p_k} f_i(\vec{x}, t) \quad x_{i,p_k}(0) = 0$$

While the sensitivity function $x_{i,p_k}(t)$ is a general measure for the change of state $x_i(t)$ with respect to parameter p_k, the *Normalized Sensitivity Function* $\lambda_{i,p_k}(t)$ measures quantitatively the change of the state $x_i(t)$ due to a 1% relative change of the parameter p_k:

$$\lambda_{i,p_k}(t) = x_{i,p_k}(t) \cdot \frac{p_k}{100}$$

The nonlinear pendulum with two states and four parameters deduces eight sensitivity functions and ODE sensitivity equations $\varphi_d, v_d, \varphi_l, v_l, \varphi_m, v_m, \varphi_g, v_g$ with e.g.:

$$\dot{\varphi} = \frac{1}{l} \cdot v \quad \dot{v} = -g \cdot sin\varphi - \frac{d}{m} \cdot v$$

$$\dot{\varphi}_d = \frac{1}{l} \cdot v_d \quad \dot{v_d} = -g \cdot cos\varphi \cdot \varphi_d - \frac{d}{m} \cdot v_d - \frac{1}{m} \cdot v$$

$$\dot{\varphi}_l = \frac{1}{l} \cdot v_l - \frac{1}{l^2} \cdot v \quad \dot{v_l} = -g \cdot cos\varphi \cdot \varphi_l - \frac{d}{m} \cdot v_l$$

Figure 14 shows all sensitivity functions for a longer time horizon, showing interesting oscillatory behaviour especially for the length sensitivity φ_l.

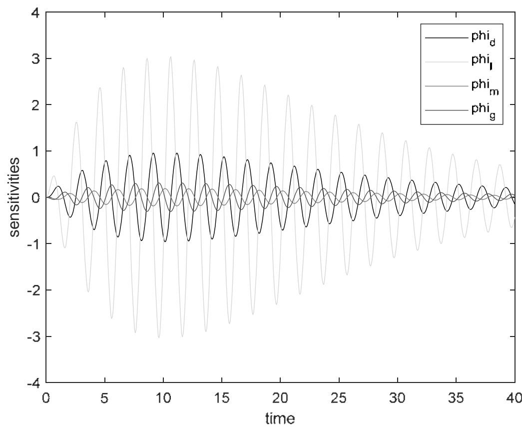

Figure 14: Sensitivity functions as ODE solutions of the sensitivity equations with additional oscillation.

The MATLAB implementation of the sensitivity system is straightforward:

```
x0 = [ phi0 v0 0 0 0 0 0 0 0 0 ]';
[t, x] = ode45(@(t,x) pend_sens(t,x,rl,rl2,dm,rm,dmm,g),...
        [0, tend], x0);
function dxdt = pend_sens(t,x,rl,rl2,dm,rm,dmm,g)
dxdt=zeros(10,1);
dxdt = [ rl*x(2);          -g*sin(x(1))      - dm*x(2);          ...
     rl*x(4);          -g*cos(x(1))*x(3) - dm*x(4)  - rm*x(2)  ;...
     rl*x(6)-rl2*x(2);  -g*cos(x(1))*x(5) - dm*x(6);          ...
     rl*x(8);          -g*cos(x(1))*x(7) - dm*x(8)  + dmm*x(2) ;...
     rl*x(10);          -g*cos(x(1))*x(9) - dm*x(10) - sin(x(1))];
end
```

Of interest for the constrained pendulum are shorter periods of the oscillations in between the events *Hit* and *Release*. Figure 15 shows the normalized sensitivity functions for length and damping: a length change is dominating:

$$\lambda_{\varphi,d}(t) = \varphi_d(t)\cdot\frac{d}{100} \quad \lambda_{\varphi,l}(t) = \varphi_d(t)\cdot\frac{l}{100}$$

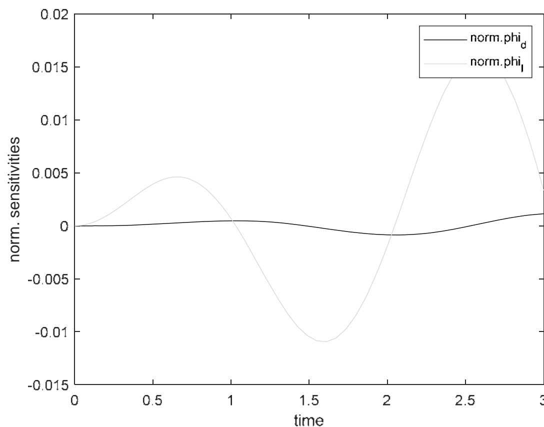

Figure 15: Normalized sensitivity functions for damping (smooth) and length (dominating).

Of course the sensitivity functions can be continued after an event, starting with nonzero initial values (final values of previous segment).

Already the small pendulum system shows that derivation of the sensitivity functions is voluminous and error-prone. Symbolic computation in combination with appropriate function handling automatizes the derivation and simulation of sensitivity equations, e.g. with the MATLAB Symbolic Toolbox:

- Definition of the symbolic system ODE function with symbolic states and parameters
- Definition of symbolic sensitivity states, symbolic derivation of the sensitivity ODE functions
- Composing system ODE functions and sensitivity ODE functions to complete symbolic function set
- Transformation of symbolic sensitivity ODEs to numerical ODE (vector) function for ODE simulation

Nevertheless, the sensitivity analysis with sensitivity functions and sensitivity ODE system is based on continuous dependency of the parameters – which is not the case for the parameter pin position φ_{pin}.

4.2 Sensitivity by Monte-Carlo Method

Partly simpler is another method for sensitivity analysis, the *Monte-Carlo Method* (used also for other tasks, as simulation itself). Generally, Monte-Carlo technique works with multiple random disturbances of inputs, calculating the multiple system responses, and finally calculating mean and standard deviation of the responses.

For the constrained pendulum this technique could be used for the pendulum parameters d, l, m, g and time courses $\varphi(t; d)$ and $\varphi(t; l)$ as system response – without benefit compared to sensitivity functions. But the technique offers itself for analyzing the sensitivity of the event times $t_{e,k}$ with respect to the pin angle φ_{pin}.

Starting with a sufficient big sample of disturbed pin angles $\varphi_{pin}^{[rand_i]}$, the resulting *Hit* and *Release* event times

$$t_{e,k}(\varphi_{pin}^{[rand_i]})$$

for k=1,…, n_e (#events), i=1,…,n_s (#samples) are calculated, and then statistically evaluated with mean value and standard deviation:

$$t_{e,k}^{mean} = \frac{1}{n_s}\sum_i t_{e,k}\left(\varphi_{pin}^{[rand_i]}\right), k = 1, \ldots n_e,$$

$$t_{e,k}^{std} = \sqrt{\frac{1}{n_s}\sum_i \left(t_{e,k}\left(\varphi_{pin}^{[rand_i]}\right) - t_{e,k}^{mean}\right)^2}$$

Sensitivity investigations will consider now the event times $t_{e,k}^{e}$ for the nonlinear model with event detection, the event times $t_{e,k}^{n}$ for the nonlinear model without event detection, and the event time differences

$$\Delta_{e,k}^{e-n} = t_{e,k}^{e} - t_{e,k}^{n}$$

An implementation in MATLAB is very easy, because calculation of mean value and standard deviation are basic tasks. Simulation is performed in a loop with sample size n_s, collecting a matrix T_e^e (t_events_e_mc) with n_e event times for each event time and for each sample:

$$T_e^e = \begin{pmatrix} t_{e,1}^{e,1} & t_{e,2}^{e,1} & \cdots & t_{e,n_e}^{e,1} \\ t_{e,1}^{e,2} & t_{e,2}^{e,2} & \cdots & t_{e,n_e}^{e,2} \\ \vdots & \vdots & \ddots & \vdots \\ t_{e,1}^{e,n_s} & t_{e,2}^{e,n_s} & \cdots & t_{e,n_e}^{e,n_s} \end{pmatrix}$$

and equivalently the matrix T_e^n (t_events_n_mc) for event times without event detection, and Δ_e^{e-n} (t_events_dif_mc) for the time differences, with results

$t_{e,k}^{e,mean}$ (mean_t_events_e), $t_{e,k}^{e,std}$ (std_t_events_e)

$t_{e,k}^{n,mean}$ (mean_t_events_n), $t_{e,k}^{n,std}$ (std_t_events_n),

$\Delta_{e,k}^{e-n,mean}$ (mean_t_events_dif), $\Delta_{e,k}^{e-n,std}$ (std_t_events_dif).

In contrary to the symbolic vector and matrix notation the implementation is very simple:

```
mean_t_events_e   = mean(t_events_e_mc)
std_t_events_e    = std(t_events_e_mc)
mean_t_events_n   = mean(t_events_n_mc)
std_t_events_n    = std(t_events_n_mc)
mean_t_events_dif = mean(t_events_dif_mc)
std_t_events_dif  = std(t_events_dif_mc)
```

Event No.	t_e^e	$t_e^{e,mean}$ / $t_e^{e,std}$	$t_{e,k}^{n,mean}$ / $t_{e,k}^{n,std}$	$\Delta_e^{e-n,mean}$ / $\Delta_{e,k}^{e-n,std}$
1	0.7034	0.7034	0.7039	0.0007
		0.0022	0.0024	0.0009
2	1.1520	1.1519	1.1522	0.0005
		0.0004	0.0007	0.0006
3	2.5901	2.5903	2.5912	0.0009
		0.0057	0.0059	0.0011
4	2.9905	2.9907	2.9917	0.0010
		0.0029	0.0031	0.0012
5	4.5425	4.5426	4.5423	0.0015
		0.0110	0.0109	0.0019
6	4.8672	4.8676	4.8675	0.0017
		0.0053	0.0056	0.0021

Table 2: Monte-Carlo sensitivity analysis for event times with and without event detection, nonlinear model.

Table 2 summarizes the results for a Monte-Carlo study with a 5% uniformly distributed change in pin position, with a sample of $n_s = 500$ tolerance of 1e-4. The results indicate that event times are not very sensitive with respect to small changes in the pin position, and that deviations are in the same range as the deviations between event times with and without state detection (the step size control in case of no event detection really seems to compensate the missing event detection).

5 Comparison of Event Detection

In case of nonlinear dynamics, the quest for the 'exact' event time t_e^{exact} cannot really be determined – all ODE solutions with (t_e^e) and without (t_e^n) event algorithms are only approximations.

Results with event detection in MATLAB, Simulink, and Stateflow are reliable and very close, if the ODE45 parameters (tolerances, maximal stepsize) are chosen properly (Table 3, case A, C, D). Because the last 'short' pendulum phase is very short (~0.045 sec), all ODE approaches must limit the stepsize.

The quick and 'dirty' approaches without event algorithm calculate astonishing results for the event times t_e^n: partly they occur before the 'exact' ones t_e^e – contrary to expectation (Table 3, Simulink case A vs. B, MATLAB case D vs E & F). Responsible is the stepsize control: the jump in length causes smaller stepsizes to keep the tolerances – in vain: step size control ends up with $t_{k+1}^n = t_e^n$ violating the tolerances (hidden warnings). The step size control seems to replace the event algorithm, although it is mathematically wrong (discontinuity). Is the earlier event time t_e^n more exact than the later t_e^e ? – no, because different solver parameters let also t_e^e happen earlier. Furthermore, the 'dubious' stepsize control lets events t_e^n vanish, which can be corrected by new solver tuning (Table 3, Simulink case B, MATLAB case E vs. F).

All linear model solutions are close to the nonlinear ones. For algorithmic event detection, linear models can make use of an ODE solver – with similar results for t_e^e and t_e^n (Table 3, case G vs. H).

Usually, linear systems are solved 'exactly' by the exponential matrix, in a recursive loop with fixed steps. Event detection can be implemented by a 'cheap' Newton step –- successful, effective and more reliable t_e^e event times with ODE (Table 3, case E vs. G). Accuracy can be improved by smaller steps around the events. Mathematicians like the analytic solution, where event detection requires solution of nonlinear equations, either by a partly symbolic Newton algorithm, or direct symbolically, all with same 'most' exact results t_e^e and extremely close to solution with the exponential matrix (Table 3, case I vs J).

Event time	A Simulink Nonlinear Model with Event Detection rtol 1e-4	B Simulink Nonlinear Model without Event Detection rtol 1e-4	C Stateflow Nonlinear Model with Event Detection rtol 1e-4	D MATLAB Nonlinear Model with Event Detection rtol 1e-4	E MATLAB Nonlinear Model without Event Detection rtol 1e-4	F MATLAB Nonlinear Model without Event Detection rtol 1e-5	G MATLAB Linear Model ODE Solver with Event Detection rtol 1e-4	H MATLAB Linear Model ODE Solver without Event Detection rtol 1e-4	I MATLAB Linear Model Solution Exponential Matrix & Onestep Newton	J MATLAB Linear Model Analytic Solution all Approches
1.	0.703454	0.70228	0.703459	0.703459	0.703327	0.703427	0.692023	0.686477	0.692018	0.692023
2.	1.151559	1.159532	1.151763	1.151771	1.160078	1.151778	1.120535	1.130859	1.120547	1.120545
3.	2.590362	2.585031	2.590407	2.590416	2.588979	2.590338	2.540851	2.510915	2.540911	2.540860
4.	2.990219	2.986587	2.990503	2.990514	2.998792	2.990403	2.931783	2.932346	2.931825	2.931800
5.	4.542716	4.54376	4.542743	4.542752	4.543987	4.542705	4.465752	4.447178	4.465854	4.465756
6.	4.867135	4.865345	4.867457	4.867471	4.874602	4.867492	4.790766	4.782015	4.790856	4.790785
7.	6.649999	-	6.648841	6.648860	-	6.648731	6.532110	-	6.532465	6.532060
8.	6.719074	-	6.720245	6.720253	-	6.720995	6.652945	-	6.652801	6.652993

Table 3: Event times t_e^e and t_e^n for all presented approaches – 'standard' initial values.

6 Integration of Approaches into MMT E-Learning Server

The *MMT E-learning System* – MMT stands for *Mathematics, Modelling and Tools* – is a tool used at the Institute of Analysis and Scientific Computing and at TU Vienna for education in modelling and simulation (and also by other institutes dealing with education in modelling and simulation).

The MMT server, developed since 2006 ([5]), plays a major role in lectures for modelling and simulation and courses in applied mathematics. The case studies and examples for modelling and simulation deal with different kinds of modelling, like ODEs, cellular automata or agent-based models, and distinct applications.

The MMT System is a web application with

- a frontend presenting interactive examples and case studies for modelling and simulation, as well as related lecture notes,
- with MATLAB running as simulation engine,
- and with a backend content management system for preparing examples, case studies etc. and lecture note content.

Figure 16: Entrance Webpage of the MMT Server.

When a student enters the website, he is welcomed by Adam Ries (1492 – 1559; a German mathematician), with login (Figure 16).

After login, all courses the student takes are listed on the left hand side. When the student chooses a course and in this course an example, the web page offers experimentation features (Figure 17). On the left hand side a navigation lets select different examples, and further course topics (here 'Pendulum Identification').

Each course topic includes appropriate examples (Figure 18, selected 'Nonlinear Pendulum with Zero Crossing Data'). On the right hand side there is a link to the source code of the example ('view m-file'; MATLAB m-files). Furthermore there can be other files linked, which can be downloaded like lecture notes, tables, etc.

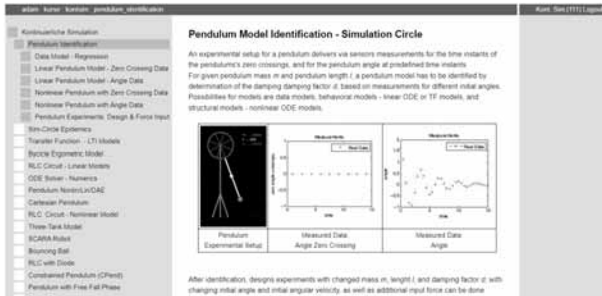

Figure 17: MMT Course Entrance.

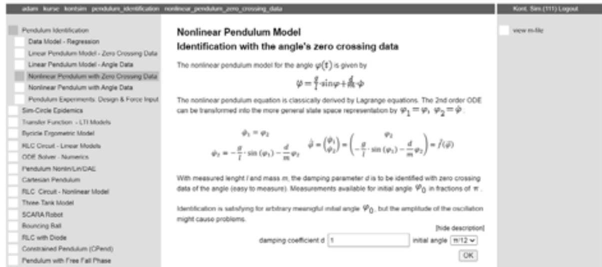

Figure 18: MMT Course Example.

The availability of the source code is an important feature of the MMT system: students can use the code for further development. In the centre information on the selected example is shown, and parameters for experiments can be chosen. With a click on the "OK" button, the server computes the results with the chosen parameters (Figure 19).

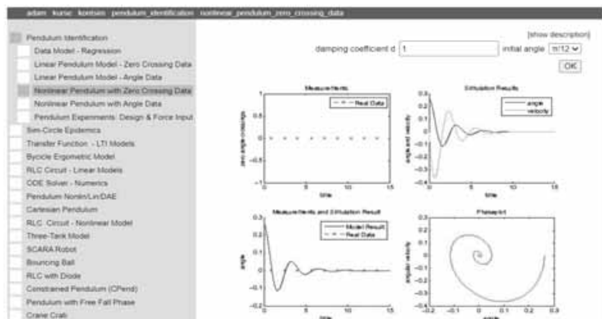

Figure 19: MMT Course Example – Results.

There are various aspects for choosing examples and case studies for the MMT server: modelling topics, applications, methods, etc. The *ARGESIM Benchmarks for Modelling Approaches and Simulation Implementations* are a challenging mixture of modelling approaches and application, and have therefore become also a basis for education in modelling and simulation ([6]), using the various solutions published in SNE. Consequently the benchmarks are also interesting topics as case studies for the MMT server, and the MMT development team has started to integrate the some benchmarks into the MMT server, taking a well-elaborated MATLAB approach with the defined tasks as examples, and extending them by further topics and tasks.

After C9 *Two-Tank Fuzzy Control*, C11 *SCARA Robot*, C12 *Collision of Spheres*, C13 *Crane with Control*, C15 *Kidney Clearance Identification*, and C17 *SIR-type Epidemics*, now C7 *Constrained Pendulum* is integrated.

The C7 integration into the MMT server (Figure 20, MMT introduction page for Constrained Pendulum) extends the benchmark tasks by topics presented in this contribution: waiving event detection, event handling methods, linear system cases, linear analytic and symbolic solutions, sensitivity analysis, and Monte-Carlo sensitivity. Additionally, the team prepares as further topic the approximation of the nonlinear model by a sequence of linear affine models with adaptive linearization points.

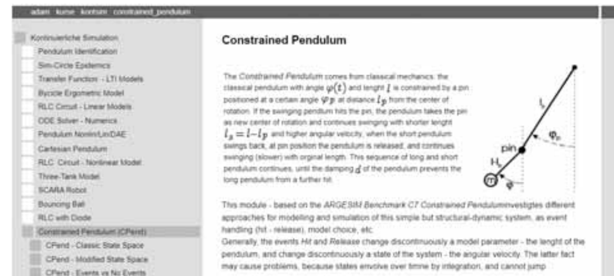

Figure 20: MMT Case Study 'Constrained Pendulum'.

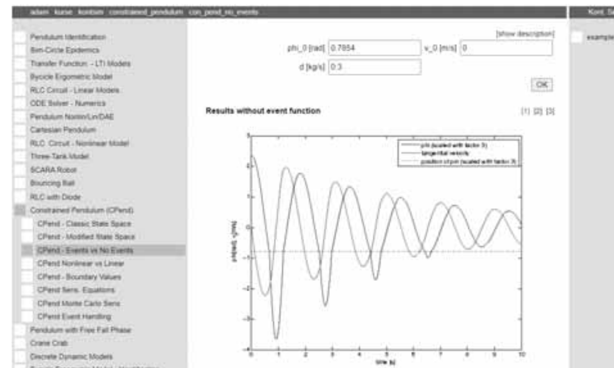

Figure 21: MMT 'Constrained Pendulum' – Results 1.

Figure 22: MMT 'Constrained Pendulum' – Results 2.

Choosing for instance the experiment 'CPend – Events vs. No Events' offers to enter model parameters. Pressing 'OK', the MATLAB engine runs, and gives back various results, as e.g. the time courses (Figure 21), or the event times (Figure 22).

At present the MMT Server offers about 200 case studies, each with various detailed examples. In winter 2021 the MMT has also extended the already integrated C17 *SIR-type Epidemic* by model identification, and lockdown and vaccination strategies based on Corona epidemics data from Austria (Figure 23).

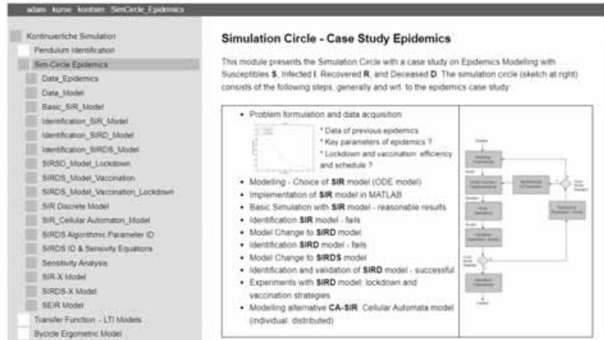

Figure 23: MMT SIR Case Study: Spread of Infection.

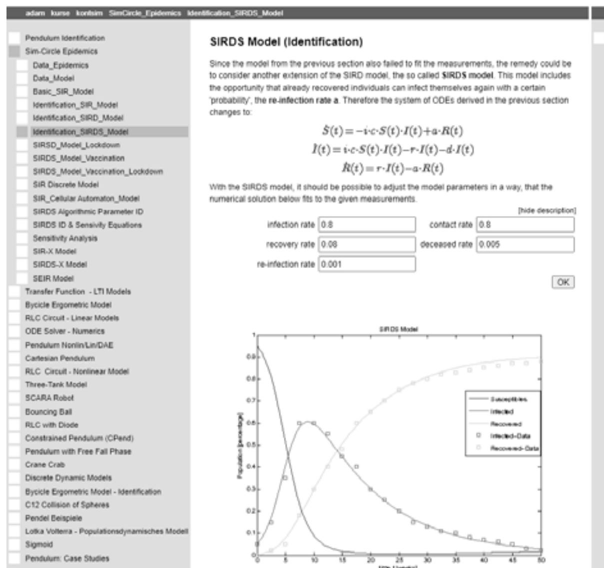

Figure 24: SIR Case Study – Identification.

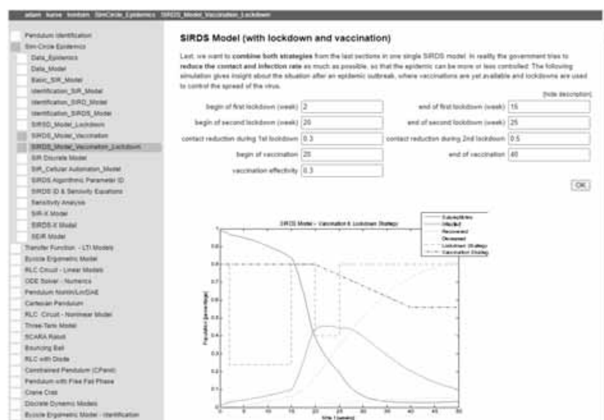

Figure 25: SIR Case Study – Lockdown and Vaccination.

In some simulation courses, students could investigate with the MMT Server strategies again pandemics (Figure 24 and Figure 25).

And last but not least, a view into the MMT backend with the model interface page for the constrained pendulum (Figure 26). Novices need about one week to learn how to work and develop in this backend.

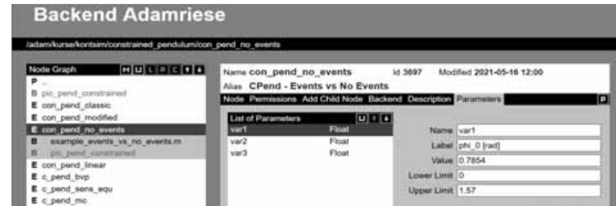

Figure 26: MMT Case Study 'Constrained Pendulum' – Backend with parameter interface.

Acknowledgement. The development of new MMT case studies is also done by students themselves – they present their seminar work, their practical course, or parts of their bachelor work or diploma work also in the MMT server – to be used by other students. This contribution on the benchmark *Constrained Pendulum* with all extensions and the MMT integration is result of a student project work from the lecture 'Continuous Simulation', composed by four students from mechatronics, and by the supervising tutor and lecturer, who added theoretical topics and MMT preparation.

References

[1] Breitenecker F. Comparison 7: Constrained Pendulum, Definition. SNE 3(7), 1993, 29

[2] Körner A, Breitenecker F. State Events and Structural-dynamic Systems: Definition of ARGESIM Benchmark C21. Simulation Notes Europe SNE 26(2), 2016, 117 – 122. DOI: 10.11128/sne.26.bn21.10339

[3] Stockinger AE, Gütl E, Rath S, Strasser D, Bicher M, Körner A, Ecker H. Direct Implementation of ARGESIM Benchmark C7 'Constrained Pendulum' in MATLAB and EXCEL. SNE 29(2), 2019, 105-110. DOI: 10.11128/sne.29.bne07.10478

[4] Moler C, Van Loan C. Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later. SIAM Review 45(1):3-49. DOI: 10.1137/S00361445024180

[5] Winkler S, Körner A, Popper N. MMT – Mathematics, Modelling and Tools: An E-Learning Environment for Modelling and Simulation. SNE 21(2), 2011, 99-102. DOI: 10.11128/sne.21.en.10069

[6] Breitenecker F, Körner A, Ecker H, Popper N, Pawletta T. ARGESIM Benchmarks on Modelling Approaches and Simulation Implementations – Development, Classification and Basis for Simulation Education. SNE 29(1), 2019, 49-61. DOI: 10.11128/sne.29.bn.10468