Numerical Simulation of Dynamic Systems: Hw11 - Problem

Prof. Dr. François E. Cellier
Department of Computer Science
ETH Zurich

May 14, 2013

[H9.1] Runge-Kutta-Fehlberg with Root Solver

In homework problem [H7.1], we have implemented a Runge-Kutta-Fehlberg algorithm with Gustaffsson step-size control.

[H9.1] Runge-Kutta-Fehlberg with Root Solver

In homework problem [H7.1], we have implemented a Runge-Kutta-Fehlberg algorithm with Gustaffsson step-size control.

In this new homework, we wish to augment that code with a *root solver for handling state events* and an *event calendar* for handling time events.

[H9.1] Runge-Kutta-Fehlberg with Root Solver

In homework problem [H7.1], we have implemented a Runge-Kutta-Fehlberg algorithm with Gustaffsson step-size control.

In this new homework, we wish to augment that code with a *root solver for handling state events* and an *event calendar* for handling time events.

To this end, you are to code a **Matlab** function:

function $[y, xc, xd, tout] = \text{rkf45rt}(xc0, xd0, t, tol)$

[H9.1] Runge-Kutta-Fehlberg with Root Solver

In homework problem [H7.1], we have implemented a Runge-Kutta-Fehlberg algorithm with Gustaffsson step-size control.

In this new homework, we wish to augment that code with a *root solver for handling state events* and an *event calendar* for handling time events.

To this end, you are to code a **Matlab** function:

```
function [y, xc, xd, tout] = rkf45rt(xc0, xd0, t, tol)
```

where x_{c0} is a column vector containing the initial values of the continuous state variables; x_{d0} is a column vector containing the initial values of the discrete state variables; t is a row vector of communication instants in time; and tol is the desired absolute error bound on the states and also on the zero-crossing functions.

[H9.1] Runge-Kutta-Fehlberg with Root Solver

In homework problem [H7.1], we have implemented a Runge-Kutta-Fehlberg algorithm with Gustaffsson step-size control.

In this new homework, we wish to augment that code with a *root solver for handling state events* and an *event calendar* for handling time events.

To this end, you are to code a **Matlab** function:

function $[y, xc, xd, tout] = \text{rkf45rt}(xc0, xd0, t, tol)$

where x_{c0} is a column vector containing the initial values of the continuous state variables; x_{d0} is a column vector containing the initial values of the discrete state variables; t is a row vector of communication instants in time; and tol is the desired absolute error bound on the states and also on the zero-crossing functions.

The function returns y, a matrix of output values, where each row denotes one output variable, and each column denotes one time instant, at which the output variables were recorded; x_c is the matrix of continuous state variables; x_d is the matrix of discrete state variables; and $tout$ is the vector of time instants, at which the states and outputs were recorded.

[H9.1] Runge-Kutta-Fehlberg with Root Solver II

tout is the same as *t*, but augmented by event times. Each event time gets logged twice, once just before the event, and once just after the event.

[H9.1] Runge-Kutta-Fehlberg with Root Solver II

tout is the same as *t*, but augmented by event times. Each event time gets logged twice, once just before the event, and once just after the event.

Function *rkf45rt* calls upon a number of internal functions:

[H9.1] Runge-Kutta-Fehlberg with Root Solver II

tout is the same as *t*, but augmented by event times. Each event time gets logged twice, once just before the event, and once just after the event.

Function *rkf45rt* calls upon a number of internal functions:

- A single step of the Runge-Kutta-Fehlberg algorithm is being computed by the function:

  ```
  function [xc4, xc5] = rkf45rt_step(xc, xd, t, h)
  ```

  which looks essentially like the routine you coded earlier. x_d is treated like a parameter vector, since the discrete state variables don't change their values except at event times.

[H9.1] Runge-Kutta-Fehlberg with Root Solver III

- We check on zero-crossings using the function:

  function $[iter] \;= \text{zc\_iter}(f, tol)$

  where f is a matrix with two column vectors. The first column vector contains the values of the zero-crossing functions at the beginning of the interval, and the second column vector contains the values of the zero-crossing functions at the end of the interval. tol is the largest distance from zero, for which the iteration will terminate.

[H9.1] Runge-Kutta-Fehlberg with Root Solver III

- We check on zero-crossings using the function:

  function $[iter] = \text{zc\_iter}(f, tol)$

  where f is a matrix with two column vectors. The first column vector contains the values of the zero-crossing functions at the beginning of the interval, and the second column vector contains the values of the zero-crossing functions at the end of the interval. tol is the largest distance from zero, for which the iteration will terminate.

  The variable $iter$ returns 0, if no zero crossing occurred in the interval; it returns $+1$, if either multiple zero crossings took place inside the interval, or if a single zero crossing took place that hasn't converged yet; it returns $-i$, if one zero crossing took place and has converged. The index i is the index of the zero-crossing function that triggered the state event.

[H9.1] Runge-Kutta-Fehlberg with Root Solver IV

- If *iter* $= 1$, we wish to perform one iteration step of *regula falsi*. To this end, we code the function:

  function [*tnew*] = reg_falsi(t, f)

  where t is a row vector of length two containing the time values corresponding to the beginning and the end of the interval, respectively, and f is the same matrix used also by function *zc_iter*.

[H9.1] Runge-Kutta-Fehlberg with Root Solver IV

- If *iter* $= 1$, we wish to perform one iteration step of *regula falsi*. To this end, we code the function:

  function $[tnew] = \text{reg\_falsi}(t, f)$

  where t is a row vector of length two containing the time values corresponding to the beginning and the end of the interval, respectively, and f is the same matrix used also by function *zc_iter*.

  The variable t_{new} returns the time instant inside the interval, at which the model is to be evaluated next.

[H9.1] Runge-Kutta-Fehlberg with Root Solver IV

- If $iter = 1$, we wish to perform one iteration step of *regula falsi*. To this end, we code the function:

  function $[tnew] = \text{reg\_falsi}(t, f)$

  where t is a row vector of length two containing the time values corresponding to the beginning and the end of the interval, respectively, and f is the same matrix used also by function *zc_iter*.

  The variable t_{new} returns the time instant inside the interval, at which the model is to be evaluated next.

  The *reg_falsi* routine needs to take care of intervals containing a single triggered zero-crossing function or multiple triggered zero-crossing functions.

[H9.1] Runge-Kutta-Fehlberg with Root Solver V

The *event calendar* is maintained in a *global variable*, called *evt_cal*.

[H9.1] Runge-Kutta-Fehlberg with Root Solver V

The *event calendar* is maintained in a *global variable*, called *evt_cal*.

evt_cal is a matrix with two columns. Each row specifies one time event. The left entry denotes the event time, whereas the right entry denotes the event type, a positive integer.

[H9.1] Runge-Kutta-Fehlberg with Root Solver V

The *event calendar* is maintained in a *global variable*, called *evt_cal*.

evt_cal is a matrix with two columns. Each row specifies one time event. The left entry denotes the event time, whereas the right entry denotes the event type, a positive integer.

The events are time-ordered. The next event is always stored in the top row of the *evt_cal* matrix.

[H9.1] Runge-Kutta-Fehlberg with Root Solver V

The *event calendar* is maintained in a *global variable*, called *evt_cal*.

evt_cal is a matrix with two columns. Each row specifies one time event. The left entry denotes the event time, whereas the right entry denotes the event type, a positive integer.

The events are time-ordered. The next event is always stored in the top row of the *evt_cal* matrix.

Since this class concerns itself with *continuous systems simulation* and not with *discrete event simulation*, we shall implement the event calendar in a simple straight-forward manner as a matrix, rather than as a linear forward and backward linked list.

[H9.1] Runge-Kutta-Fehlberg with Root Solver V

The *event calendar* is maintained in a *global variable*, called *evt_cal*.

evt_cal is a matrix with two columns. Each row specifies one time event. The left entry denotes the event time, whereas the right entry denotes the event type, a positive integer.

The events are time-ordered. The next event is always stored in the top row of the *evt_cal* matrix.

Since this class concerns itself with *continuous systems simulation* and not with *discrete event simulation*, we shall implement the event calendar in a simple straight-forward manner as a matrix, rather than as a linear forward and backward linked list.

The event calendar is maintained by three functions: *push_evt*, *pull_evt*, and *query_evt*.

[H9.1] Runge-Kutta-Fehlberg with Root Solver VI

- The function:

  function push_evt(*t*, *evt_nbr*)

  inserts a time event in the event calendar in the appropriate position.

[H9.1] Runge-Kutta-Fehlberg with Root Solver VI

- The function:

  function push_evt(t, evt_nbr)

  inserts a time event in the event calendar in the appropriate position.

- The function:

  function [$tnext$, evt_nbr] = pull_evt()

  extracts the next time event from the event calendar.

[H9.1] Runge-Kutta-Fehlberg with Root Solver VI

- The function:

  function push_evt(t, *evt_nbr*)

  inserts a time event in the event calendar in the appropriate position.

- The function:

  function [*tnext*, *evt_nbr*] = pull_evt()

  extracts the next time event from the event calendar.

- The function:

  function [*tnext*, *evt_nbr*] = query_evt()

  returns the event information of the next time event without removing the event from the event calendar.

[H9.1] Runge-Kutta-Fehlberg with Root Solver VII

The model itself is stored in four different functions that the user will need to code for each discontinuous model that he or she wishes to simulate.

[H9.1] Runge-Kutta-Fehlberg with Root Solver VII

The model itself is stored in four different functions that the user will need to code for each discontinuous model that he or she wishes to simulate.

- The function:

  function $[xcdot] = \text{cst\_eq}(xc, xd, t)$

  assumes the same role that the function *st_eq* had assumed earlier. It computes the continuous state derivatives at time t. Since the discrete states x_d are constant during each continuous simulation segment, this vector assumes the role of a parameter vector.

[H9.1] Runge-Kutta-Fehlberg with Root Solver VII

The model itself is stored in four different functions that the user will need to code for each discontinuous model that he or she wishes to simulate.

- The function:

  function [*xcdot*] = cst_eq(*xc*, *xd*, *t*)

  assumes the same role that the function *st_eq* had assumed earlier. It computes the continuous state derivatives at time t. Since the discrete states x_d are constant during each continuous simulation segment, this vector assumes the role of a parameter vector.

- The function:

  function [*y*] = out_eq(*xc*, *xd*, *t*)

  assumes the same role as earlier.

[H9.1] Runge-Kutta-Fehlberg with Root Solver VII

The model itself is stored in four different functions that the user will need to code for each discontinuous model that he or she wishes to simulate.

- The function:

  function $[xcdot]$ $=$ cst_eq(xc, xd, t)

  assumes the same role that the function *st_eq* had assumed earlier. It computes the continuous state derivatives at time t. Since the discrete states x_d are constant during each continuous simulation segment, this vector assumes the role of a parameter vector.

- The function:

  function $[y]$ $=$ out_eq(xc, xd, t)

  assumes the same role as earlier.

- The new function:

  function $[f]$ $=$ zcf(xc, xd, t)

  returns the current values of the zero-crossing functions as a column vector.

[H9.1] Runge-Kutta-Fehlberg with Root Solver VIII

- The new function:

  function $[xdnew] = \text{dst\_eq}(xc, xd, t, evt\_nbr)$

  returns the new discrete state vector after an event has taken place.

[H9.1] Runge-Kutta-Fehlberg with Root Solver VIII

- The new function:

  function $[xdnew] = \text{dst\_eq}(xc, xd, t, evt\_nbr)$

  returns the new discrete state vector after an event has taken place.

  The routine handles both *time events* and *state events*. It is called with a positive event number for time events, and with a negative event number for state events.

[H9.1] Runge-Kutta-Fehlberg with Root Solver VIII

- The new function:

  function $[xdnew] \;=$ dst_eq(xc, xd, t, evt_nbr)

  returns the new discrete state vector after an event has taken place.

  The routine handles both *time events* and *state events*. It is called with a positive event number for time events, and with a negative event number for state events.

  In the case of time events, the event number distinguishes between different types of events, whereas in the case of state events, it identifies the zero-crossing function that triggered the event.

[H9.1] Runge-Kutta-Fehlberg with Root Solver VIII

- The new function:

  function $[xdnew] = \text{dst\_eq}(xc, xd, t, evt\_nbr)$

  returns the new discrete state vector after an event has taken place.

  The routine handles both *time events* and *state events*. It is called with a positive event number for time events, and with a negative event number for state events.

  In the case of time events, the event number distinguishes between different types of events, whereas in the case of state events, it identifies the zero-crossing function that triggered the event.

  In the case of a time event, the *rkf45rt* function logs the current states, then removes the time event from the event calendar, then calls function *dst_eq*, and finally logs the new states once again.

[H9.1] Runge-Kutta-Fehlberg with Root Solver VIII

- The new function:

  function $[xdnew] = \text{dst\_eq}(xc, xd, t, evt\_nbr)$

  returns the new discrete state vector after an event has taken place.

  The routine handles both *time events* and *state events*. It is called with a positive event number for time events, and with a negative event number for state events.

  In the case of time events, the event number distinguishes between different types of events, whereas in the case of state events, it identifies the zero-crossing function that triggered the event.

  In the case of a time event, the *rkf45rt* function logs the current states, then removes the time event from the event calendar, then calls function *dst_eq*, and finally logs the new states once again.

  Consequently, the *dst_eq* function does not need to remove the current time event from the event calendar, but it needs to schedule future time events that are a consequence of the current event action.

[H9.1] Runge-Kutta-Fehlberg with Root Solver IX

- The main program calculates the values of both the continuous and the discrete initial states, and it places the initial time events on the event calendar.

[H9.1] Runge-Kutta-Fehlberg with Root Solver IX

- The main program calculates the values of both the continuous and the discrete initial states, and it places the initial time events on the event calendar.
- It then calls routine $rkf45rt$ to perform the simulation.

[H9.1] Runge-Kutta-Fehlberg with Root Solver IX

- The main program calculates the values of both the continuous and the discrete initial states, and it places the initial time events on the event calendar.
- It then calls routine *rkf45rt* to perform the simulation.
- It finally plots the simulation results.

[H9.7] Thyristor

We wish to implement the thyristor-controlled train engine model, or at least a circuit very similar to the one shown in class.

[H9.7] Thyristor

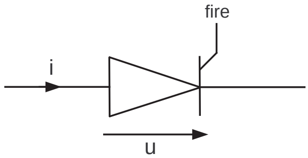
We wish to implement the thyristor-controlled train engine model, or at least a circuit very similar to the one shown in class.

The thyristor element is shown below:

[H9.7] Thyristor

We wish to implement the thyristor-controlled train engine model, or at least a circuit very similar to the one shown in class.

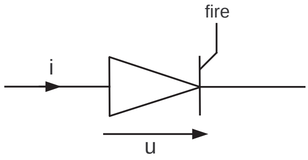
The thyristor element is shown below:



[H9.7] Thyristor

We wish to implement the thyristor-controlled train engine model, or at least a circuit very similar to the one shown in class.

The thyristor element is shown below:

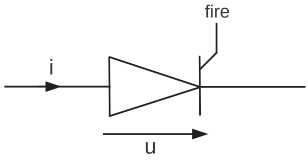


The thyristor is a diode with a modified firing logic. The diode can only close when the external Boolean variable *fire* has a value of *true*. The opening logic is the same as for the regular diode.

[H9.7] Thyristor

We wish to implement the thyristor-controlled train engine model, or at least a circuit very similar to the one shown in class.

The thyristor element is shown below:



The thyristor is a diode with a modified firing logic. The diode can only close when the external Boolean variable *fire* has a value of *true*. The opening logic is the same as for the regular diode.

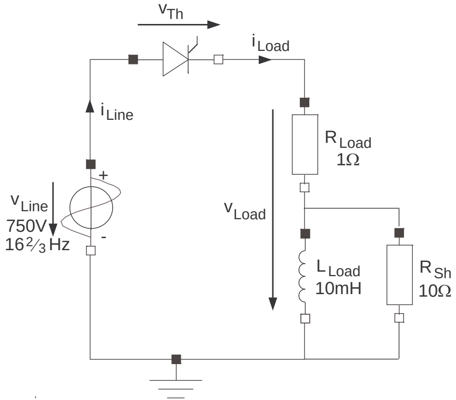
Since the thyristor is a diode, we can use the same *parameterized curve description* that we used for the regular diode. Only the switching condition is modified.

[H9.7] Thyristor II

The modified thyristor-controlled train engine model is shown below:

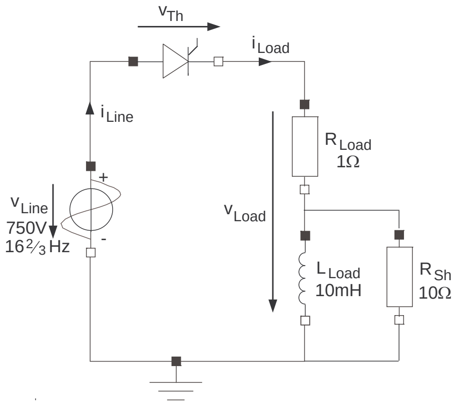
[H9.7] Thyristor II

The modified thyristor-controlled train engine model is shown below:



[H9.7] Thyristor II

The modified thyristor-controlled train engine model is shown below:



A shunt resistor was added to avoid having to deal with a *variable structure model*.

[H9.7] Thyristor III

Convert all **if**-statements of the thyristor model to their algebraic equivalents.

[H9.7] Thyristor III

Convert all **if**-statements of the thyristor model to their algebraic equivalents.

Write down all of the equations governing the thyristor-controlled rectifier circuit.

[H9.7] Thyristor III

Convert all **if**-statements of the thyristor model to their algebraic equivalents.

Write down all of the equations governing the thyristor-controlled rectifier circuit.

Draw the structure digraph of the resulting equation system and show that the switch equations indeed appear inside an algebraic loop.

[H9.7] Thyristor III

Convert all **if**-statements of the thyristor model to their algebraic equivalents.

Write down all of the equations governing the thyristor-controlled rectifier circuit.

Draw the structure digraph of the resulting equation system and show that the switch equations indeed appear inside an algebraic loop.

Choose a suitable tearing structure, and solve the equations both horizontally and vertically using the variable substitution technique.

[H9.7] Thyristor IV

Using the integration algorithms of homework problem [H9.1], simulate the model in **Matlab** across 0.2 seconds of simulated time.

[H9.7] Thyristor IV

Using the integration algorithms of homework problem [H9.1], simulate the model in **Matlab** across 0.2 seconds of simulated time.

Choose a suitable tearing structure, and solve the equations both horizontally and vertically using the variable substitution technique.

[H9.7] Thyristor IV

Using the integration algorithms of homework problem [H9.1], simulate the model in **Matlab** across 0.2 seconds of simulated time.

Choose a suitable tearing structure, and solve the equations both horizontally and vertically using the variable substitution technique.

The external control variable of the thyristor, *fire*, is to be assigned a value of *true* from the angle of 30^o until the angle of 45^o, and from the angle of 210^o until the angle of 225^o during each period of the line voltage, v_{Line}. During all other times, it is set to *false*.

[H9.7] Thyristor IV

Using the integration algorithms of homework problem [H9.1], simulate the model in **Matlab** across 0.2 seconds of simulated time.

Choose a suitable tearing structure, and solve the equations both horizontally and vertically using the variable substitution technique.

The external control variable of the thyristor, *fire*, is to be assigned a value of *true* from the angle of 30^o until the angle of 45^o, and from the angle of 210^o until the angle of 225^o during each period of the line voltage, v_{Line}. During all other times, it is set to *false*.

Plot the load voltage, v_{Load}, as well as the load current, i_{Load}, as functions of time.

[H9.7] Thyristor V

The model contains two types of *time events* that control the activation (firing) and deactivation of the thyristor control signal.

[H9.7] Thyristor V

The model contains two types of *time events* that control the activation (firing) and deactivation of the thyristor control signal.

Both an activation event (after 30^o) and a deactivation event (after 45^o) are scheduled in the initial section of the main program. Subsequent time events of the same types are scheduled always 180^o into the future as part of the event handling.

[H9.7] Thyristor V

The model contains two types of *time events* that control the activation (firing) and deactivation of the thyristor control signal.

Both an activation event (after 30^o) and a deactivation event (after 45^o) are scheduled in the initial section of the main program. Subsequent time events of the same types are scheduled always 180^o into the future as part of the event handling.

The event handling sets a discrete (Boolean) state variable, m_1, to either *true* or *false*.

[H9.7] Thyristor V

The model contains two types of *time events* that control the activation (firing) and deactivation of the thyristor control signal.

Both an activation event (after 30^o) and a deactivation event (after 45^o) are scheduled in the initial section of the main program. Subsequent time events of the same types are scheduled always 180^o into the future as part of the event handling.

The event handling sets a discrete (Boolean) state variable, m_1, to either *true* or *false*.

In **Matlab**, Booleans are represented by integers, whereby *true* $\Rightarrow 1$ and *false* $\Rightarrow 0$.

[H9.7] Thyristor VI

The model contains one *zero-crossing function*, $f = s$.

[H9.7] Thyristor VI

The model contains one *zero-crossing function*, $f = s$.

The corresponding event handling code toggles the value of another discrete (Boolean) state variable, m_s.

[H9.7] Thyristor VI

The model contains one *zero-crossing function*, $f = s$.

The corresponding event handling code toggles the value of another discrete (Boolean) state variable, m_s.

In **Matlab**, Boolean operators have been defined for the pseudo-Boolean variables in the form of functions. Thus, toggling a Boolean variable can be written as:

$ms = \text{not}(ms);$

[H9.7] Thyristor VII

The state-space model references a third discrete (Boolean) state variable, m_0.

[H9.7] Thyristor VII

The state-space model references a third discrete (Boolean) state variable, m_0.

m_0 is a Boolean function of m_1, m_s, and its own past value $\text{pre}(m_0)$. Because of the dependence of m_0 on its own past, also m_0 is a discrete state variable.

[H9.7] Thyristor VII

The state-space model references a third discrete (Boolean) state variable, m_0.

m_0 is a Boolean function of m_1, m_s, and its own past value $\mathrm{pre}(m_0)$. Because of the dependence of m_0 on its own past, also m_0 is a discrete state variable.

m_0 needs to be updated at the end of every discrete event.