Numerical Simulation of Dynamic Systems: Hw11 - Solution

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

May 28, 2013
In homework problem [H7.1], we have implemented a Runge-Kutta-Fehlberg algorithm with Gustaffsson step-size control.
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.
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:

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

```matlab
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.
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:

```matlab
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.

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.
Runge-Kutta-Fehlberg with Root Solver II

\textit{t\textsubscript{out}} is the same as \textit{t}, but augmented by event times. Each event time gets logged twice, once just before the event, and once just after the event.
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 \( \text{rkf45rt} \) calls upon a number of internal functions:
**t.out** 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:

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

  which looks essentially like the routine you coded earlier. **xd** is treated like a parameter vector, since the discrete state variables don’t change their values except at event times.
We check on zero-crossings using the function:

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

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.
We check on zero-crossings using the function:

\[
\text{function } [\text{iter}] = \text{zc_iter}(f, \text{tol})
\]

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. \( \text{tol} \) is the largest distance from zero, for which the iteration will terminate.

The variable \( \text{iter} \) 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.
Problem 9.1

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

```matlab
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 $\text{zc_iter}$. 
If \( \text{iter} = 1 \), we wish to perform one iteration step of \textit{regula falsi}. To this end, we code the function:

\begin{verbatim}
function [tnew] = reg_falsi(t, f)
\end{verbatim}

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 \textit{zc_iter}.

The variable \( t_{\text{new}} \) returns the time instant inside the interval, at which the model is to be evaluated next.
If \( \text{iter} = 1 \), we wish to perform one iteration step of *regula falsi*. To this end, we code the function:

\[
\text{function } [t_{\text{new}}] = \text{reg_falsi}(t, f)
\]

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 \( \text{zc_iter} \).

The variable \( t_{\text{new}} \) 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.
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.
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.
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 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: *pushEvt*, *pullEvt*, and *queryEvt*. 
The function:

```python
function push_evt(t, evt_nbr)
```

inserts a time event in the event calendar in the appropriate position.
The function:

```matlab
function push_evt(t, evt_nbr)
```

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

The function:

```matlab
function [tnext, evt_nbr] = pull_evt()
```

extracts the next time event from the event calendar.
The function:

```matlab
function push_evt(t, evt_nbr)
```

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

The function:

```matlab
function [tnext, evt_nbr] = pull_evt()
```

extracts the next time event from the event calendar.

The function:

```matlab
function [tnext, evt_nbr] = query_evt()
```

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

```matlab
function [dot] = 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 `xd` are constant during each continuous simulation segment, this vector assumes the role of a parameter vector.
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:**
  
  ```matlab
  function [xdot] = 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 `xd` are constant during each continuous simulation segment, this vector assumes the role of a parameter vector.

- **The function:**
  
  ```matlab
  function [y] = out_eq(xc, xd, t)
  ```

  assumes the same role as earlier.
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 [xdot] = cseq(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.
The new function:

```matlab
function [xdnew] = dst_eq(xc, xd, t, evt_nbr)
```

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

```matlab
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.
The new function:

```matlab
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.
The new function:

```matlab
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.

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.
The new function:

```matlab
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.

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.
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.
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 \textit{rkg45rt} 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 \textit{rkf45rt} to perform the simulation.

▶ It finally plots the simulation results.
The code is self-documentary. Since its parts have been explained in much detail already, there is no need to offer more explanations here.
We wish to implement the thyristor-controlled train engine model, or at least a circuit very similar to the one shown in class.
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:
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:
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.
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.
The modified thyristor-controlled train engine model is shown below:
[H9.7] Thyristor II

The modified thyristor-controlled train engine model is shown below:
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.
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.
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.
Using the integration algorithms of homework problem [H9.1], simulate the model in Matlab across 0.2 seconds of simulated time.
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.
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° until the angle of 45°, and from the angle of 210° until the angle of 225° during each period of the line voltage, vLine. During all other times, it is set to false.
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° until the angle of 45°, and from the angle of 210° until the angle of 225° during each period of the line voltage, \( v_{\text{Line}} \). During all other times, it is set to false.

Plot the load voltage, \( v_{\text{Load}} \), as well as the load current, \( i_{\text{Load}} \), as functions of time.
The model contains two types of *time events* that control the activation (firing) and deactivation of the thyristor control signal.
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°) and a deactivation event (after 45°) are scheduled in the initial section of the main program. Subsequent time events of the same types are scheduled always 180° 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°) and a deactivation event (after 45°) are scheduled in the initial section of the main program. Subsequent time events of the same types are scheduled always 180° 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*. 
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°) and a deactivation event (after 45°) are scheduled in the initial section of the main program. Subsequent time events of the same types are scheduled always 180° 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* ⇒ 1 and *false* ⇒ 0.
[H9.7] Thyristor VI

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

\[
ms = \text{not}(ms);
\]
The state-space model references a third discrete (Boolean) state variable, $m_0$. 
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.
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.

$m_0$ needs to be updated at the end of every discrete event.
[H9.7] Thyristor VIII

Homework 11 - Solution

Thyristor
[H9.7] Thyristor VIII

1: \( V_{\text{Line}} = V_0 \cdot \sin \left( \frac{2\pi t}{t_p} \right) \)
2: \( V_{\text{RLoad}} = R_{\text{Load}} \cdot i_{\text{Load}} \)
3: \( V_{\text{RSh}} = L_{\text{Load}} \cdot \frac{di_L}{dt} \)
4: \( V_{\text{RSh}} = R_{\text{Sh}} \cdot i_{\text{RSh}} \)
5: \( V_{\text{Load}} = V_{\text{RLoad}} + V_{\text{RSh}} \)
6: \( V_{\text{Line}} = V_{\text{Th}} + V_{\text{Load}} \)
7: \( i_{\text{Load}} = i_L + i_{\text{RSh}} \)
8: \( V_{\text{Th}} = m_0 \cdot s \)
9: \( i_{\text{Load}} = (1 - m_0) \cdot s \)
[H9.7] Thyristor VIII

1: \[ v_{\text{Line}} = V_0 \cdot \sin \left( \frac{2\pi t}{t_p} \right) \]
2: \[ v_{\text{RLoad}} = R_{\text{Load}} \cdot i_{\text{Load}} \]
3: \[ v_{\text{RSh}} = L_{\text{Load}} \cdot \frac{di_L}{dt} \]
4: \[ v_{\text{RSh}} = R_{\text{Sh}} \cdot i_{\text{RSh}} \]
5: \[ v_{\text{Load}} = v_{\text{RLoad}} + v_{\text{RSh}} \]
6: \[ v_{\text{Line}} = v_{\text{Th}} + v_{\text{Load}} \]
7: \[ i_{\text{Load}} = i_L + i_{\text{RSh}} \]
8: \[ v_{\text{Th}} = m_0 \cdot s \]
9: \[ i_{\text{Load}} = (1 - m_0) \cdot s \]

\( m_0 \) is a discrete state variable. It is true, when the thyristor is off.
Numerical Simulation of Dynamic Systems: Hw11 - Solution

Homework 11 - Solution

**Thyristor**

[H9.7] Thyristor IX

1: \( v_{\text{Line}} = V_0 \cdot \sin\left(\frac{2\pi t}{t_p}\right) \)

2: \( v_{\text{RLoad}} = R_{\text{Load}} \cdot i_{\text{Load}} \)

3: \( v_{\text{RSh}} = L_{\text{Load}} \cdot \frac{di}{dt} \)

4: \( v_{\text{RSh}} = R_{\text{Sh}} \cdot i_{\text{RSh}} \)

5: \( v_{\text{Load}} = v_{\text{RLoad}} + v_{\text{RSh}} \)

6: \( v_{\text{Line}} = v_{\text{Th}} + v_{\text{Load}} \)

7: \( i_{\text{Load}} = i_L + i_{\text{RSh}} \)

8: \( v_{\text{Th}} = m_0 \cdot s \)

9: \( i_{\text{Load}} = (1 - m_0) \cdot s \)
1: \( v_{\text{Line}} = V_0 \cdot \sin\left(\frac{2\pi t}{t_p}\right) \)

2: \( v_{\text{RLoad}} = R_{\text{Load}} \cdot i_{\text{Load}} \)

3: \( v_{\text{RSh}} = L_{\text{Load}} \cdot \frac{di_{\text{L}}}{dt} \)

4: \( v_{\text{RSh}} = R_{\text{Sh}} \cdot i_{\text{RSh}} \)

5: \( v_{\text{Load}} = v_{\text{RLoad}} + v_{\text{RSh}} \)

6: \( v_{\text{Line}} = v_{\text{Th}} + v_{\text{Load}} \)

7: \( i_{\text{Load}} = i_{\text{L}} + i_{\text{RSh}} \)

8: \( v_{\text{Th}} = m_0 \cdot s \)

9: \( i_{\text{Load}} = (1 - m_0) \cdot s \)
[H9.7] Thyristor X

We causalize as much as we can:
We causalize as much as we can:

\[ \text{Eq.}(1) \quad v_{\text{Line}} \]

\[ \text{Eq.}(9) \quad v_{\text{RLoad}} \]

\[ \text{Eq.}(\quad) \quad i_{\text{Load}} \]

\[ \text{Eq.}(\quad) \quad v_{\text{RSh}} \]

\[ \text{Eq.}(\quad) \quad i_{\text{RSh}} \]

\[ \text{Eq.}(\quad) \quad v_{\text{Load}} \]

\[ \text{Eq.}(\quad) \quad \frac{di_{\text{L}}}{dt} \]

\[ \text{Eq.}(\quad) \quad v_{\text{Th}} \]

\[ \text{Eq.}(\quad) \quad s \]
We causalize as much as we can:

1: \( v_{\text{Line}} = V_0 \cdot \sin\left(\frac{2\pi t}{t_p}\right) \)

?: \( v_{\text{RLoad}} = R_{\text{Load}} \cdot i_{\text{Load}} \)

9: \( v_{\text{RSh}} = L_{\text{Load}} \cdot \frac{di_{\text{L}}}{dt} \)

?: \( v_{\text{RSh}} = R_{\text{Sh}} \cdot i_{\text{RSh}} \)

?: \( v_{\text{Load}} = v_{\text{RLoad}} + v_{\text{RSh}} \)

?: \( v_{\text{Line}} = v_{\text{Th}} + v_{\text{Load}} \)

?: \( i_{\text{Load}} = i_{\text{L}} + i_{\text{RSh}} \)

?: \( v_{\text{Th}} = m_0 \cdot s \)

?: \( i_{\text{Load}} = (1 - m_0) \cdot s \)
We causalize as much as we can:

\[ v_{\text{Line}} = V_0 \cdot \sin\left(\frac{2\pi t}{t_p}\right) \]
\[ v_{\text{RLoad}} = R_{\text{Load}} \cdot i_{\text{Load}} \]
\[ v_{\text{RSh}} = L_{\text{Load}} \cdot \frac{di_L}{dt} \]
\[ v_{\text{RSh}} = R_{\text{Sh}} \cdot i_{\text{RSh}} \]
\[ v_{\text{Load}} = v_{\text{RLoad}} + v_{\text{RSh}} \]
\[ v_{\text{Line}} = v_{\text{Th}} + v_{\text{Load}} \]
\[ i_{\text{Load}} = i_L + i_{\text{RSh}} \]
\[ v_{\text{Th}} = m_0 \cdot s \]
\[ i_{\text{Load}} = (1 - m_0) \cdot s \]

We end up with an algebraic loop in seven equations and seven unknowns. The switch equation (variable \( s \)) is part of the loop.
We choose $s$ as our first tearing variable:
We choose $s$ as our first tearing variable:
We choose $s$ as our first tearing variable:

1: $v_{\text{Line}} = V_0 \cdot \sin\left(\frac{2\pi t}{t_p}\right)$

?: $v_{\text{RLoad}} = R_{\text{Load}} \cdot i_{\text{Load}}$

9: $v_{\text{RSh}} = L_{\text{Load}} \cdot \frac{di_L}{dt}$

?: $v_{\text{RSh}} = R_{\text{Sh}} \cdot i_{\text{RSh}}$

?: $v_{\text{Load}} = v_{\text{RLoad}} + v_{\text{RSh}}$

3: $v_{\text{Line}} = v_{\text{Th}} + v_{\text{Load}}$

?: $i_{\text{Load}} = i_L + i_{\text{RSh}}$

2: $v_{\text{Th}} = m_0 \cdot s$

res.eq.1: $i_{\text{Load}} = (1 - m_0) \cdot s$
We choose $s$ as our first tearing variable:

1: $v_{Line} = V_0 \cdot \sin \left(\frac{2\pi t}{t_p}\right)$

?: $v_{RLoad} = R_{Load} \cdot i_{Load}$

9: $v_{RSh} = L_{Load} \cdot \frac{di_L}{dt}$

?: $v_{RSh} = R_{Sh} \cdot i_{RSh}$

?: $v_{Load} = v_{RLoad} + v_{RSh}$

3: $v_{Line} = v_{Th} + v_{Load}$

?: $i_{Load} = i_L + i_{RSh}$

2: $v_{Th} = m_0 \cdot s$

res.eq.1: $i_{Load} = (1 - m_0) \cdot s$

We end up with a second algebraic loop in four equations and four unknowns.
We choose a second residual equation, and now, we can causalize the remaining equations:
We choose a second residual equation, and now, we can causalize the remaining equations:

1. Eq.(1)
2. Eq.(2)
3. Eq.(3)
4. Eq.(4)
5. Eq.(5)
6. Eq.(6)
7. Eq.(9)
8. Res.Eq.1
9. Res.Eq.2

These equations are connected with variables such as: vLine, vRLoad, vRSh, iLoad, iRSh, vLoad, diL/dt, vTh, and s.
We choose a second residual equation, and now, we can causalize the remaining equations:

1: \( v_{\text{Line}} = V_0 \cdot \sin\left(\frac{2\pi t}{t_p}\right) \)
2: \( v_{\text{Th}} = m_0 \cdot s \)
3: \( v_{\text{Load}} = v_{\text{Line}} - v_{\text{Th}} \)
4: \( v_{\text{RSh}} = v_{\text{Load}} - v_{\text{RLoad}} \)
5: \( i_{\text{RSh}} = \frac{1}{R_{\text{Sh}}} \cdot v_{\text{RSh}} \)
6: \( i_{\text{Load}} = i_L + i_{\text{RSh}} \)
7: \( s = 1 - m_0 \cdot i_{\text{Load}} \)
8: \( \frac{di_{\text{L}}}{dt} = \frac{1}{L_{\text{Load}}} \cdot v_{\text{RSh}} \)
Substitution gives us two linear equations in the two unknown tearing variables, $s$ and $v_{RLoad}$:

\[ [R_{Sh} \cdot (1 - m_0) + m_0] \cdot s + v_{RLoad} = R_{Sh} \cdot i_L + v_{Line} \]
\[ (m_0 \cdot R_{Load}) \cdot s + (R_{Load} + R_{Sh}) \cdot v_{RLoad} = (R_{Load} \cdot R_{Sh}) \cdot i_L + R_{Load} \cdot v_{Line} \]
Substitution gives us two linear equations in the two unknown tearing variables, $s$ and $v_{RLoad}$:

$$[R_{Sh} \cdot (1 - m_0) + m_0] \cdot s + v_{RLoad} = R_{Sh} \cdot i_L + v_{Line}$$

$$m_0 \cdot R_{Load} \cdot s + (R_{Load} + R_{Sh}) \cdot v_{RLoad} = (R_{Load} \cdot R_{Sh}) \cdot i_L + R_{Load} \cdot v_{Line}$$

or:

$$\begin{pmatrix} R_{Sh} \cdot (1 - m_0) + m_0 & 1 \\ m_0 \cdot R_{Load} & R_{Load} + R_{Sh} \end{pmatrix} \cdot \begin{pmatrix} s \\ v_{RLoad} \end{pmatrix} = \begin{pmatrix} R_{Sh} & 1 \\ R_{Load} \cdot R_{Sh} & R_{Load} \end{pmatrix} \cdot \begin{pmatrix} i_L \\ v_{Line} \end{pmatrix}$$
Substitution gives us two linear equations in the two unknown tearing variables, $s$ and $v_{RLoad}$:

\[
\begin{align*}
[R_{Sh} \cdot (1 - m_0) + m_0] \cdot s + v_{RLoad} &= R_{Sh} \cdot i_L + v_{Line} \\
(m_0 \cdot R_{Load}) \cdot s + (R_{Load} + R_{Sh}) \cdot v_{RLoad} &= (R_{Load} \cdot R_{Sh}) \cdot i_L + R_{Load} \cdot v_{Line}
\end{align*}
\]

or:

\[
\begin{pmatrix}
R_{Sh} \cdot (1 - m_0) + m_0 \\
m_0 \cdot R_{Load}
\end{pmatrix}
\begin{pmatrix}
1 \\
R_{Load} + R_{Sh}
\end{pmatrix}
\begin{pmatrix}
s \\
v_{RLoad}
\end{pmatrix}
= \begin{pmatrix}
R_{Sh} \\
R_{Load} \cdot R_{Sh}
\end{pmatrix}
\begin{pmatrix}
1 \\
R_{Load}
\end{pmatrix}
\begin{pmatrix}
i_L \\
v_{Line}
\end{pmatrix}
\]

We are now ready to code.
function [xcdot] = cst_eq(xc, xd, t)
    
    % State - space model of [H9.7]
    
    RLoad = 1;
    RSh = 10;
    LLoad = 0.01;
    V0 = 750;
    p = 16 + 2/3;
    tp = 1/p;
    
    iL = xc(1);
    m0 = xd(1);
    m1 = xd(2);
    ms = xd(3);
    
    vLine = V0*sin(2 * pi * t / tp);
    inp = [iL; vLine];
    a11 = RSh * (1 - m0) + m0;
    a12 = 1;
    a21 = m0 * RLoad;
    a22 = RLoad + RSh;
    A = [a11, a12; a21, a22];
    b11 = RSh;
    b12 = 1;
    b21 = RLoad * RSh;
    b22 = RLoad;
    B = [b11, b12; b21, b22];
    tear = A \ B * inp;
    s = tear(1);
    vRLoad = tear(2);
    vTh = m0 * s;
    vLoad = vLine - vTh;
    vRSh = vLoad - vRLoad;
    iRSh = vRSh / RSh;
    iLoad = iL + iRSh;
    diL = vRSh / LLoad;
    
    xcdot = diL;
    
    return
function [y] = out_eq(xc, xd, t)
    % Output model of [H9.7]
    RLoad = 1;
    RSh = 10;
    LLoad = 0.01;
    V0 = 750;
    p = 16 + 2/3;
    tp = 1/p;
    iL = xc(1);
    m0 = xd(1);
    m1 = xd(2);
    ms = xd(3);
    vLine = V0*sin(2*pi*t/tp);
    inpt = [iL; vLine];
    a11 = RSh*(1-m0) + m0;
    a12 = 1;
    a21 = m0*RLoad;
    a22 = RLoad + RSh;
    A = [a11, a12; a21, a22];
    b11 = RSh;
    b12 = 1;
    b21 = RLoad*RSh;
    b22 = RLoad;
    B = [b11, b12; b21, b22];
    tear = A\B * inpt;
    s = tear(1);
    vRLoad = tear(2);
    vTh = m0*s;
    vLoad = vLine - vTh;
    vRSh = vLoad - vRLoad;
    iRSh = vRSh/RSh;
    iLoad = iL + iRSh;
    diL = vRSh/LLoad;
    y = zeros(2, 1);
    y(1) = vLoad;
    y(2) = iLoad;
    return
function \[ f \] = zcf(xc, xd, t)
\%
% Zero - crossing function of [H9.7]
\%
RLoad = 1;
RSh = 10;
LLoad = 0.01;
V0 = 750;
p = 16 + 2/3;
\%
\%
iL = xc(1);
m0 = xd(1);
m1 = xd(2);
ms = xd(3);
\%
vLine = V0*sin(2 * pi * t / tp);
inpt = [iL; vLine];
a11 = RSh * (1 - m0) + m0;
a12 = 1;
a21 = m0 * RLoad;
a22 = RLoad + RSh;
A = [a11, a12; a21, a22];
b11 = RSh;
b12 = 1;
b21 = RLoad * RSh;
b22 = RLoad;
B = [b11, b12; b21, b22];
tear = A\B * inpt;
s = tear(1);
vRLoad = tear(2);
vTh = m0 * s;
vLoad = vLine - vTh;
vRSh = vLoad - vRLoad;
iRSh = vRSh/RSh;
iLoad = iL + iRSh;
diL = vRSh/LLoad;
\%
f = s;
\%
return
We still need to discuss the *thyristor logic*. Let us check how the Modelica Standard Library (MSL) tackles the problem:
We still need to discuss the *thyristor logic*. Let us check how the **Modelica Standard Library (MSL)** tackles the problem:
We still need to discuss the *thyristor logic*. Let us check how the Modelica Standard Library (MSL) tackles the problem:

The MSL uses a *leaky diode*.
[H9.7] Thyristor XVIII

```model IdealThyristor "Ideal thyristor"
  extends Modelica.Electrical.Analog.Interfaces.OnePort;
  parameter Modelica.SIunits.Resistance Ron(final min=0) = 1.E-5;
  "Closed thyristor resistance";
  parameter Modelica.SIunits.Conductance Goff(final min=0) = 1.E-5;
  "Opened thyristor conductance";
  parameter Modelica.SIunits.Voltage Vknee(final min=0, start=0);
  "Forward threshold voltage";
  Boolean off(start=true) "Switching state";

  protected
    Real s(final unit="1")
    "Auxiliary variable: if on then current, if opened then voltage";
    constant Modelica.SIunits.Voltage unitVoltage= 1 s;
    constant Modelica.SIunits.Current unitCurrent= 1 s;

  equation
    off = s < 0 or pre(off) and not fire;
    v = (s*unitVoltage)*(if off then 1 else Ron) + Vknee;
    i = (s*unitVoltage)*(if off then Goff else 1) + Goff*Vknee;
end IdealThyristor;
```
Using our ideal diode:

\[
\begin{align*}
  \text{off} &= s < 0 \text{ or pre}(\text{off}) \text{ and not fire}; \\
  v_{Th} &= \text{if off then } s \text{ else } 0; \\
  i_{Load} &= \text{if off then } 0 \text{ else } s;
\end{align*}
\]
Using our ideal diode:

\[ \text{off} = s < 0 \text{ or pre}(\text{off}) \text{ and not fire}; \]
\[ v_{Th} = \text{if off then } s \text{ else } 0; \]
\[ i_{Load} = \text{if off then } 0 \text{ else } s; \]

or in terms of our variables:

\[ m_s = s < 0; \]
\[ m_0 = m_s \text{ or pre}(m_0) \text{ and not } m_1; \]
\[ v_{Th} = \text{if } m_0 \text{ then } s \text{ else } 0; \]
\[ i_{Load} = \text{if } m_0 \text{ then } 0 \text{ else } s; \]
Using our ideal diode:

\[
\text{off} = s < 0 \text{ or pre}(\text{off}) \text{ and not fire};
\]
\[
\nu_{Th} = \text{if } \text{off} \text{ then } s \text{ else } 0;
\]
\[
\nu_{Load} = \text{if } \text{off} \text{ then } 0 \text{ else } s;
\]

or in terms of our variables:

\[
m_s = s < 0;
\]
\[
m_0 = m_s \text{ or pre}(m_0) \text{ and not } m_1;
\]
\[
\nu_{Th} = \text{if } m_0 \text{ then } s \text{ else } 0;
\]
\[
\nu_{Load} = \text{if } m_0 \text{ then } 0 \text{ else } s;
\]

and using \textbf{Matlab’s} pseudo-Boolean variables and functions:

\[
m_0_{\text{new}} = \text{or}(m_s, \text{and}(m_0, \text{not}(m_1)));
\]
function [xdnew] = dst_eq(xc, xd, t, evt_nbr)

    p = 16 + 2/3;
    tp = 1/p;
    il = xc(1);
    m0 = xd(1);
    m1 = xd(2);
    ms = xd(3);
    if evt_nbr == 1,
        m1 = 1;
        push_evt(t + tp/2, 1);
    end
    if evt_nbr == 2,
        m1 = 0;
        push_evt(t + tp/2, 2);
    end

    if evt_nbr == -1,
        ms = not(ms);
    end

    iL = xc(1);
    m0 = xd(1);
    m1 = xd(2);
    ms = xd(3);
    if evt_nbr == 1,
        m1 = 1;
        push_evt(t + tp/2, 1);
    end
    if evt_nbr == 2,
        m1 = 0;
        push_evt(t + tp/2, 2);
    end

    if evt_nbr == -1,
        ms = not(ms);
    end

    x = zeros(3, 1);
    x(1) = m0;
    x(2) = m1;
    x(3) = ms;
    return
[H9.7] Thyristor XXI