9

Simulation of Discontinuous Systems

Preview

In this chapter, we shall discuss how discontinuous models can be handled by the simulation software, and in particular by the numerical integration algorithm. Discontinuous models are extremely common in many areas of engineering, e.g. to describe dry friction phenomena or impact between bodies in mechanical engineering, or to describe switching circuits in electronics. In the first part of this chapter, we shall be dealing with the numerical aspects of integrating across discontinuities. Two types of discontinuities are introduced, time events and state events, that require different treatment by the simulation software. In the second part of this chapter, we shall discuss the modeling aspects of how discontinuities can be conveniently described by the user in an object–oriented manner, and what the compiler needs to do to translate these object–oriented descriptions down into event descriptions.

9.1 Introduction

As we have seen, all numerical integration algorithms used in today's simulation programs are based, either explicitly or implicitly, on Taylor–Series expansions. Simulation trajectories are always approximated by polynomials or rational functions in the step size h around the current time t_k.

This causes problems when dealing with discontinuous models, since polynomials never exhibit discontinuities at all, and also rational functions only exhibit occasional poles, but no discontinuities. Thus, if an integration algorithm tries to integrate across a discontinuity, it will invariably be in trouble.

Since the step size is finite, the integration algorithm doesn't recognize a discontinuity as such. It simply notices that the trajectory suddenly and unexpectedly changes its behavior by showing symptoms of a very steep gradient. Thus, the integration algorithm experiences the discontinuity as the sudden appearance of a new eigenvalue far out to the left in the complex plane. If the algorithm is step–size controlled, it will react to this observation by reducing the step–size in order to shrink the eigenvalue into the asymptotic region of the $(\lambda \cdot h)$–plane. Unfortunately, this new eigenvalue

has the nasty habit of being evasive. Although the step size is made smaller and smaller, the eigenvalue doesn't allow itself to be captured. The integration algorithm thus experiences the discontinuity as a *singular point of infinite stiffness*.

The algorithm finally gives up, as its step size is either reduced to the smallest tolerable value, or because the step–size control is getting fooled. We shall see why this can easily happen. As a consequence, the discontinuity is passed through with a very small step size ... and the spooky phenomenon vanishes as fast as it appeared. The integration algorithm notices that the funny eigenvalue has disappeared again, and consequently will enhance the step size in the steps to come, until the appropriate optimal step size has been regained. It is in this fashion that the step–size control within the numerical integration algorithm is able to handle discontinuities ... and often, it does so with quite decent success.

Figure 9.1 illustrates how step–size control handles discontinuities.

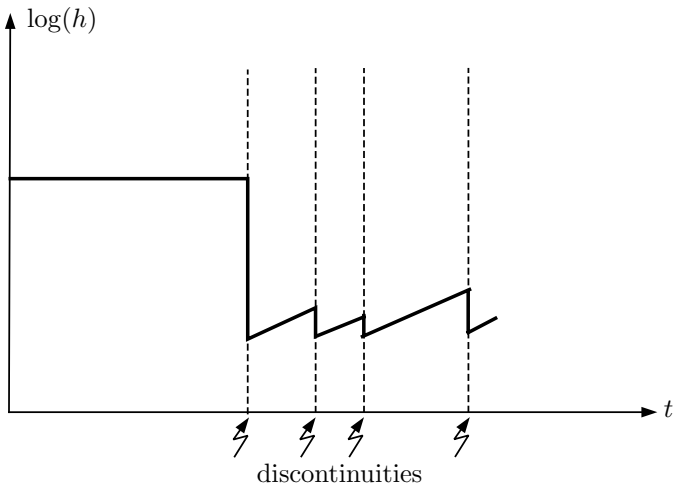

FIGURE 9.1. Discontinuity handling by step–size control.

Figure 9.1 shows the logarithm of the step size, h, plotted across simulated time, t. As the integration algorithm approaches a discontinuity, the step size is reduced until the algorithm judges the solution to be correct. After the discontinuity has passed, the step size is cautiously increased again until the next discontinuity is encountered. This is quite inefficient, but often produces decent results.

It is this lucky by–product of the step–size control mechanism that allowed the simulation software producers to get by for many years without spending too much of a thought on the problem of discontinuity handling. Unfortunately, things can go awfully awry as was demonstrated in [9.5].

9.2 Basic Difficulties

In the seventies, one of the authors was a Ph.D. student at ETH Zürich in Switzerland. He was working on a dissertation on exactly the topic of this chapter [9.5]. One day, a colleague of his, who had difficulties with his simulation program, came to see him. He had worked on his program for weeks and weeks, and it simply didn't want to run properly. He was another Ph.D. student, working on the design of a velocity controller for electrically driven locomotive engines [9.25]. When analyzing his friend's problem, he soon realized that his program exhibited difficulties that were closely related to the way the numerical integration algorithm handled the discontinuities in his model. Let us explain.

In Switzerland, electric train engines are operated by AC current with a frequency of $16\frac{2}{3}$ Hz. The amplitude of the voltage available to the engine is constant, thus velocity control cannot be achieved by simply modifying the voltage. An Ohmic voltage divider is out of the question, since we want to propel the engine, not heat it up. Variable transformers, on the other hand, are too large and bulky.

Previously, train engines in Switzerland had been equipped with a thyristor circuit controlling the firing angle of the thyristor. Figure 9.2 shows the circuit diagram of the thyristor circuit.

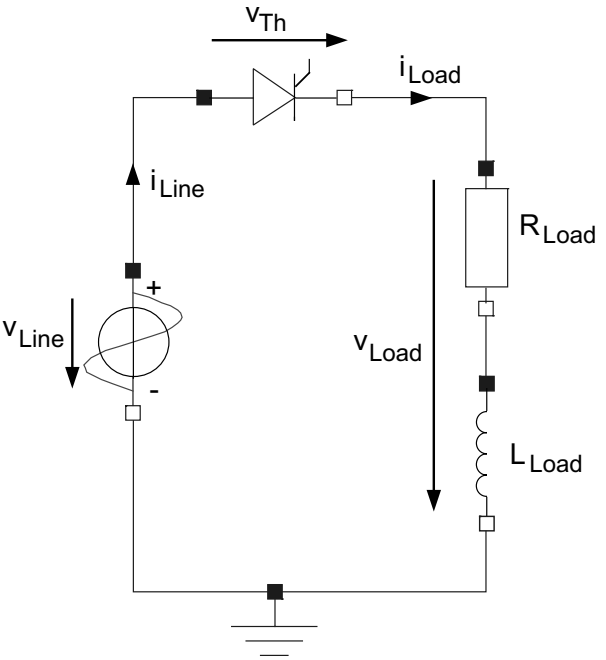

FIGURE 9.2. Circuit diagram of thyristor circuit for train speed control.

The partly Ohmic partly inductive load represents the engine. This model is simplified, but shall do to explain the difficulties with this approach. The thyristor is a switch element. It can be "fired" (i.e., closed) by applying a low voltage impulse to the thyristor gate. The thyristor then stays on until the current through the thyristor passes through zero. At zero current, the thyristor automatically opens again.

Figure 9.3 shows the current, i_{Load}, flowing through the load and the voltage, v_{Load} across the load, assuming that the thyristor is repetitively fired by an impulse applied once every period after a given firing angle α. In the example, we chose $\alpha = 30^{\text{o}}$.

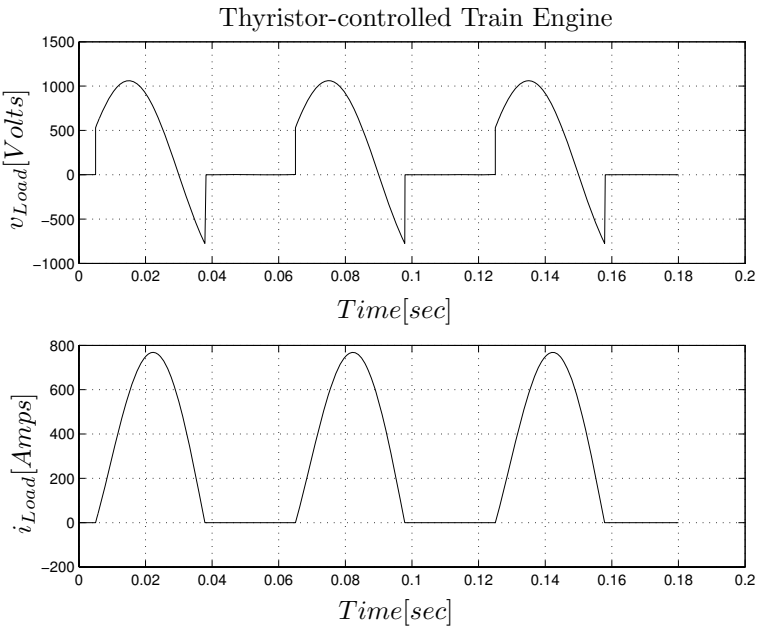

FIGURE 9.3. Voltage and current of thyristor–controlled train engine.

The Ohmic power made available to the engine for conversion to mechanical power is approximately:

$$P_{\text{Ohmic}} = v_{\text{Load}} \cdot i_{\text{Load}} \tag{9.1}$$

Evidently, it is possible to control the Ohmic power by changing the firing angle α. For $\alpha = 0^{\text{o}}$, the full sine wave goes through, i.e., the power is maximized. For $\alpha \geq 180^{\text{o}}$, no power goes through at all.

This control strategy worked exceedingly well and almost everyone was very happy ... except for the electricity company of the Canton of Uri. Let us explain.

Figure 9.4 shows the power spectrum of the thyristor–controlled voltage signal.

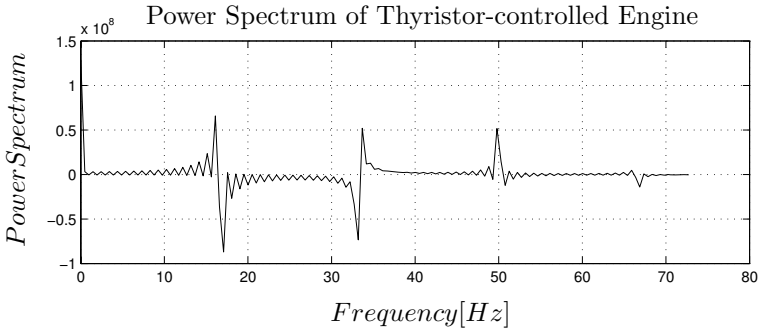

FIGURE 9.4. Power spectrum of thyristor–controlled voltage signal.

This was computed by simulating the above circuit across 1.5 seconds of simulated time using 1200 communication points. A *fast Fourier transform (FFT)* of P_{Ohmic} was then computed. Figure 9.4 shows the real part of the low frequency end of that spectrum plotted across frequency.

Roughly 17% of the power is DC power, 30% are at the base frequency, another 30% are at the 2^{nd} harmonic, roughly 15% are at the 3^{rd} harmonic, and 4% are at the 4^{th} harmonic.

The 3^{rd} harmonic thus carries a substantial percentage of the overall power of the signal. Unfortunately, the 3^{rd} harmonic happens to be located at 50 Hz, i.e., precisely at the frequency, with which the electric power company delivers electric power to the households in Switzerland. It so happened that whenever one of these trains (usually equipped with two engines) drove up the St.Gotthard mountain, the electric counters in households located near the rails were reset to zero.

Next, the train engineers tried *burst control*. Figure 9.5 shows the circuit diagram of a burst–controlled engine.

Figure 9.6 shows the voltage across and current through the train engine when using burst control. The high–voltage circuitry is very similar to the one used in the previous approach. This time, we use two thyristors with a common gate control logic. However, the gate control of the thyristor now works differently. Rather than letting through a certain percentage of every period, the burst–controlled thyristor fires constantly during a certain number of periods, and then stops firing for the remainder of the burst.

It was decided to use bursts of eight periods. Consequently, the burst frequency is one eighth of the line frequency, i.e., $2\frac{1}{12}$ Hz. Out of these eight periods, a certain number of periods is being let through, and the remainder is stopped. In Fig.9.6, five out of every eight periods are let through. Evidently, engines using this speed control strategy cannot operate at an arbitrary percentage of the full power, but only at $\frac{1}{8}^{\text{th}}$, or $\frac{2}{8}^{\text{th}}$, or $\frac{3}{8}^{\text{th}}$, etc. of the full power.

The advantages of this simple solution were twofold. On the one hand, it solved the problem of resetting the electric counters, since the power

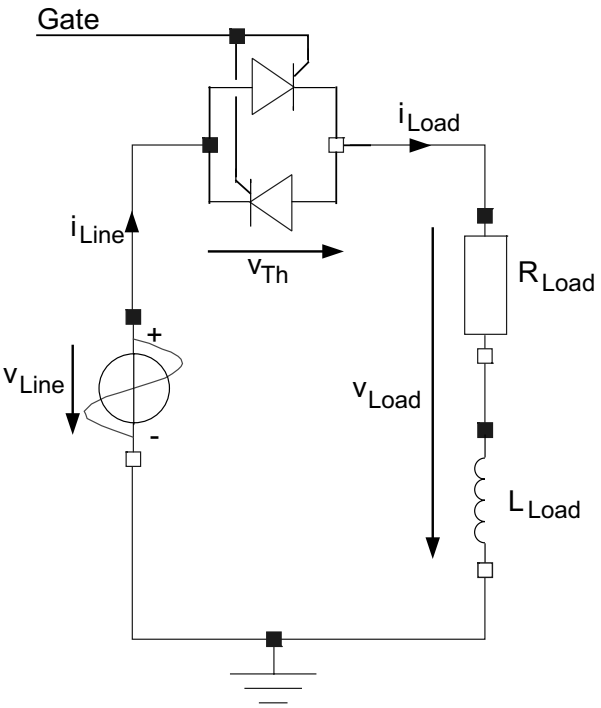

FIGURE 9.5. Circuit diagram of thyristor circuit for burst control.

spectrum no longer contains a significant amount of power at 50 Hz, and secondly, it was very cheap, since the (expensive) high–voltage circuitry needed very little modification. Only the (comparatively inexpensive) low–voltage circuitry needed to be replaced.

These circuits were installed in the trains that served the northern shore of Lake Zürich, on the line Zürich–Meilen–Rapperswil, and were used there for a number of years. When the train pulled out of the station, it operated during one burst (about 0.5 seconds) at $\frac{1}{8}^{\text{th}}$ of full power, then during the next burst at $\frac{2}{8}^{\text{th}}$, etc. These trains weren't able to accelerate smoothly. The speed changed abruptly, which the customers felt noticeably in their stomachs. It just wasn't very comfortable.

Thus, our colleague had been asked to come up with something better. He designed the circuitry shown in Fig.9.7.

This time, the engine is represented by something that drains current out of the net, i.e., as a current source. The representation is not accurate, but it is good enough for the task at hand. Also, the line frequency has been normalized to $\omega = 2\pi f = 1\ \text{sec}^{-1}$, so that the same circuit would also work for other countries with different line frequencies. The impedance values have been adjusted accordingly.

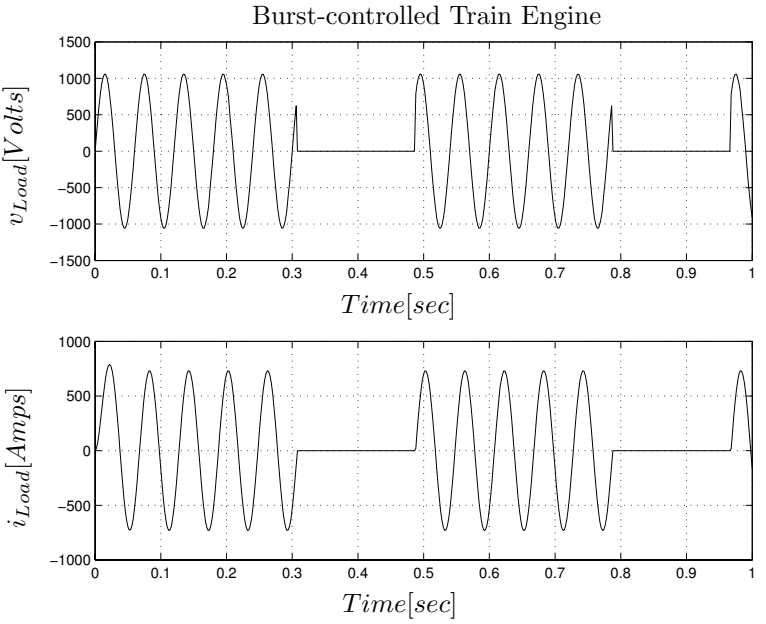

FIGURE 9.6. Voltage and current of burst–controlled train engine.

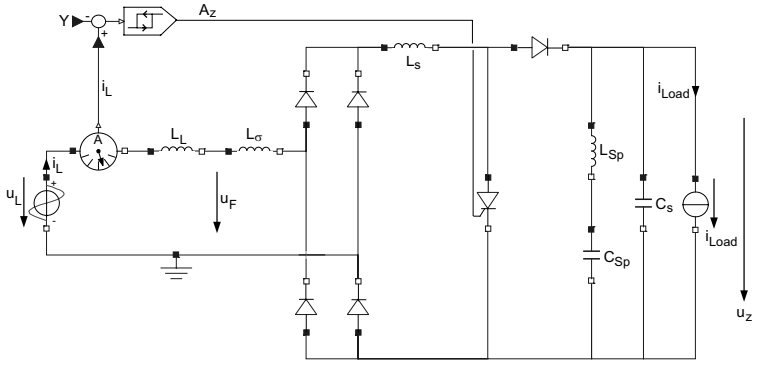

FIGURE 9.7. Circuit diagram of SCR circuit for train speed control.

The gate control logic is also shown on Fig.9.7. The line current, i_L, is controlled in such a way that it always remains in the vicinity of:

$$Y(t) = \frac{15 \cdot 10^6}{u_L} \sin \omega t \tag{9.2}$$

For $A_z = 0.0$, the line current, i_L, grows rapidly until it crosses $(Y + B_T)$ in the positive direction. At that moment, A_z assumes a value of $A_z = 1.0$, and i_L decays quickly again until it reaches $(Y - B_T)$, where A_z takes a

value of $A_Z = 0.0$ as before.

$$B_T = 200.0 \text{ Amps} \tag{9.3}$$

is the allowed tolerance around $Y(t)$, within which i_L is supposed to operate.

Figure 9.8 shows two signals of this circuit during the first half–period, namely the filter voltage u_F within the control loop, and the load voltage, u_z.

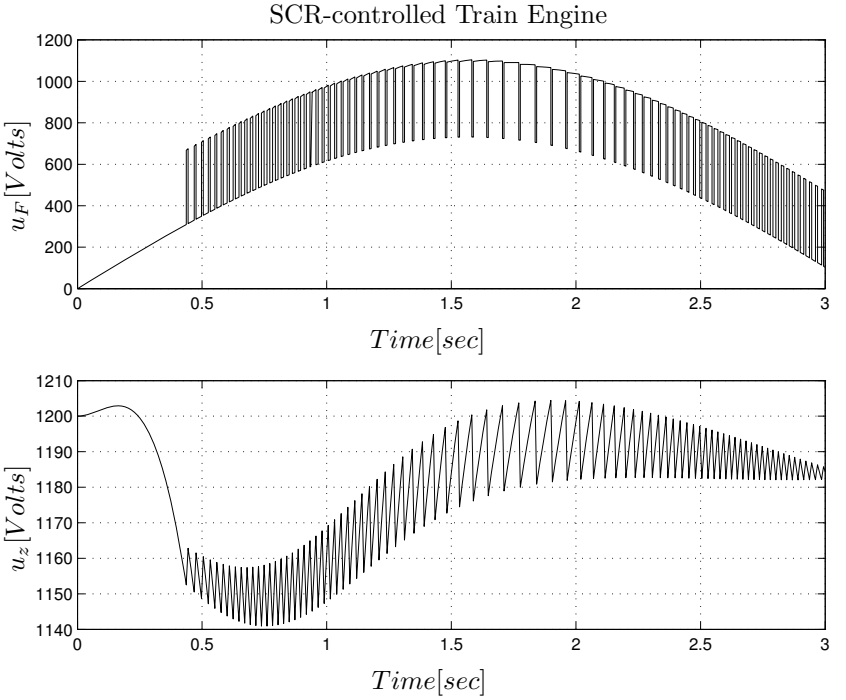

FIGURE 9.8. Filter and load voltage in SCR–controlled train engine.

If a numerical integration algorithm could fall into depression, this might be as good a reason for it to do so as any. What nightmarish curves to integrate over (!) The filter voltage, u_F, after an initial transitory phase, essentially follows a sine wave. It toggles back and forth between the sine wave itself and the same curve with a constant DC value of about 300 Volts superposed. The load voltage, u_z, is regulated to stay essentially at a constant value, in the given example somewhere around 1184 Volts. The power spectrum of the load is mostly DC, except for a small percentage located at frequencies much higher than 50 Hz.

However, these are not the results that our colleague had found, when he came to discuss his simulation results. Figure 9.9 presents the results

that he had obtained.

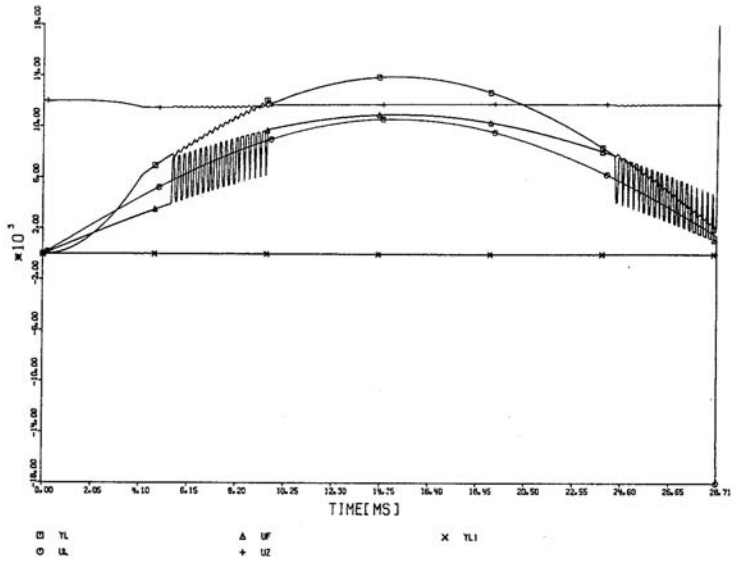

FIGURE 9.9. Filter voltage in SCR–controlled train engine.

This is an old plot that we scanned in from [9.5]. We were unable to reproduce precisely the results that our colleague had obtained, since they had been produced by an old simulation software, CSMP–III, that we don't have around any longer. It was a software specifically designed for use on IBM mainframes, machines that have been moth–balled long ago.

The graph shows the filter voltage, u_F, plotted over time together with some other signals. The reader notices that the curve looks similar to the newly obtained one, except during the time interval from about 8 msec to 24 msec, when the filter voltage on the old plot didn't exhibit the high frequency oscillation.

The simulation took forever to run. For this reason, we recommended to our friend to also plot the step size, h, used as a function of simulated time. It is shown in Figure 9.10.

The step size varies a lot over time, as the step–size control algorithm is being used to catch the discontinuities. However, it is quite evident from the plot that the simulation uses consistently a very small step size during the period from 8 msec to 24 msec, i.e., the period, during which the simulation results are incorrect. The simulation exhibits *creeping behavior*.

Somehow, the gate control had gotten stuck. The numerical integration algorithm was aware of that fact and tried to fix it by using very small step sizes, but was unable to do so. Thus, using the step–size control mechanism

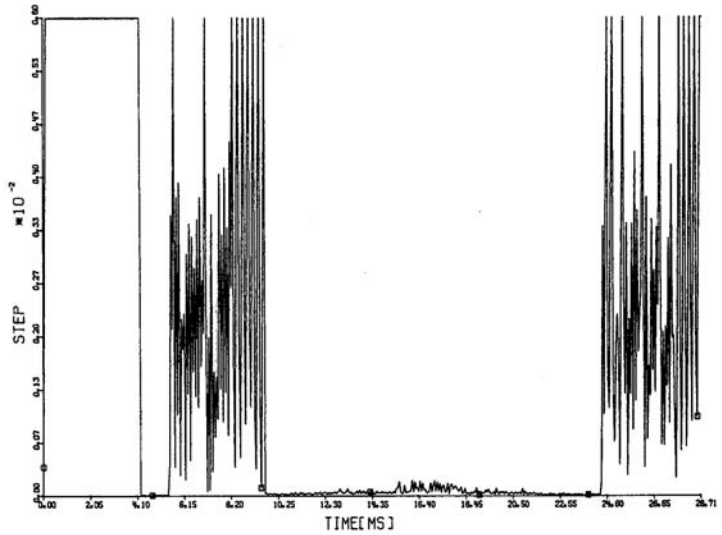

FIGURE 9.10. Step size in SCR–controlled train engine.

for handling discontinuities evidently is not only *inefficient*, it can also be *dangerous*. Notice that the simulation program did not produce any error message at all.

How can these results be explained? The step–size control mechanism of any step–size–controlled integration algorithm is based on an error estimate. This error estimate, for an n^{th}–order algorithm, is something like:

$$\varepsilon = c \cdot h^{n+1} \tag{9.4}$$

Consequently, as we reduce the step size more and more, the error estimate *will* become smaller and smaller, irrespective of whether the integration makes any sense or not. Practically speaking, as we reduce the step size, the higher–order terms in the Taylor–Series expansion become less and less important until, finally, every integration algorithm behaves like Euler. Explicit algorithms will behave like forward Euler, whereas implicit algorithms may behave either like forward Euler or like backward Euler.

If we try to integrate across a discontinuity, the two formulae that are compared to each other for the purpose of step–size control, will eventually both behave like Euler, and at that time, they will agree on their "solution" ... not necessarily the *correct* solution, mind you, but at least a solution they both came up with. If two numerical codes agree on a solution to a problem, that may indeed indicate that the solution is correct ... but it may just as well simply mean that the two codes employ the same (possibly flawed) algorithm. Therefore, if two different numerical codes miraculously

agree on a solution to almost machine resolution, we are usually much more suspicious of foul play than if their agreement is less spectacular.

Clearly, in the case of Fig.9.9, this is what happened. During some periods of time, the two algorithms agreed happily on the same –evidently quite wrong– solution. What happened was the following. The program used a step–size controlled explicit single–step algorithm, some variant of a fourth–order Runge–Kutta method, more precisely, it used the Runge–Kutta–Simpson method described in H3.16, a rather dubious method, as we now understand.

When the solution approached the threshold, the solution managed to switch several times back and forth within a single integration step. If the number of switchings happened to be *odd*, the step ended with the other model, and integration proceeded as desired. On the other hand, if the number of switchings was *even*, the step ended in the same switch position it had started out with, and the algorithm went through the same switching immediately again during the next step. This explains why the solution was creeping along the switching boundary, unable to leave it.

Abusing the step–size control for locating discontinuities is always quite inefficient. The reason is that the algorithm doesn't know, and cannot know, that a discontinuity is taking place. It must therefore assume the worst, namely that the system is highly nonlinear with rapidly changing eigenvalues of its Jacobian matrix. Consequently, the algorithm has to be cautious in increasing its step size again after the discontinuity has been cleared in order to avoid potential numerical instability problems that may be caused by a hyperactive step–size adjustment strategy. This is documented in Fig.9.1, where the step size remains constantly at too small a value since the next discontinuity is always encountered before the step size could regain its optimal value.

Abusing the step–size control for locating discontinuities can sometimes lead to incorrect results that may be difficult to identify as such, i.e., incorrect results may be produced and go unnoticed. The above application is a good example of that.

We evidently need something better.

9.3 Time Events

In many cases, we do know some time in advance when a discontinuity will take place. For example, in the case of the original thyristor gate control logic, we know that the thyristor will close exactly α° after the start of each period. It is just a question of providing this information to the integration algorithm. Discontinuities will from now on be called *discrete events*, and if we know when such an event will take place, we can *schedule* it to happen by entering the *event time* and the *event type* into a *calendar of forthcoming*

events.

The event calendar is a linearly–linked list of events arranged in the order of increasing times of occurrence, thus the first event in the event calendar is always the *next event*. In the case of multiple simultaneous events, additional tie–breaking rules can be specified to decide which event comes first. The sequence may matter. For example, if a car arrives at a traffic light that simultaneously switches to red, it may make a big difference whether the simulation program decides that the car arrived first, or whether it decides that the light changed first. Therefore, tie–breaking rules should be implemented, and should be considered carefully.

The next event time is considered by the step–size control of the integration algorithm exactly like a communication or readout point. If the integration algorithm usually will adjust the step size in the vicinity of a readout point in order to hit the point accurately (mostly done in the case of single–step algorithms), then so should it treat the next event time. If the next event time falls in between the current time and the time when the next step should ordinarily end, the step size is reduced in order not to miss the communication point. If the next event can be reached by increasing the next step by not more than 10%, then this is justifiable in order to prevent a very short step thereafter. On the other hand, if the integration algorithm interpolates in order to visit the next communication point (mostly done in multi–step integration by use of the Nordsieck vector), then it should do the same in order to accommodate the next event time.

Notice that no discontinuity takes places while the event is being located. The discontinuity is not directly coded into the model, only the condition of its occurrence is. Thus, the trajectories seen by the integration algorithm are perfectly continuous, and the integration algorithm therefore has nothing to worry about.

Once the next event time has been located, the continuous simulation comes to a halt, and a discrete event section of the simulation program is visited that implements the consequences of the event taking place, i.e., sets the state variables to their new values, changes the current values of input functions, etc. It is this section that implements the discontinuities. A simulation program may contain many different discrete event sections, one for every event type.

The end result of event handling can be considered a new set of initial conditions, from which a completely new integration can start. Thus, a simulation run across a discontinuous model can be interpreted as a sequence of distinct strictly continuous simulation runs, separated by discrete events.

The recipe is so trivial that one would assume that all serious continuous system simulation languages (CSSLs) would meanwhile have adopted it ... or faced the destiny of natural attrition. However, due to the soi–disant "event handling" capabilities of the step–size control algorithms themselves, many simulation software designers never bothered to look into the issue ... and so far they got away with it. Well, hopefully this book

will finally change all of this.

Let us consider once more the thyristor–controlled train engine model. The gate needs to be closed after α°. Thus, the time of the first time event that closes the gate takes place at:

$$t_{\text{period}} = \frac{1}{2\pi f} \tag{9.5a}$$

$$t_{\text{event}} = \frac{\alpha}{360} \cdot t_{\text{period}} \tag{9.5b}$$

Since we know from the beginning of the simulation, when this event is going to take place, the event can be scheduled in the initialization portion of the simulation program.

Thus, the *initialization section* of the simulation program could contain the statements (in pseudo–code):

Gate = *open*
schedule *CloseGate* **at** *t_event*

The *event description section* of the simulation program would then close the gate, and schedule the next gate closing event one period later:

Gate = *closed*
schedule *CloseGate* **at** t + *t_period*

The variable *Gate* can be referred to from within the continuous–time simulation model. This is not dangerous, since discrete states behave exactly like parameters or constants as far as the integration algorithm is concerned. They never change their values while the integration is proceeding. They only change their values in between segments of numerical integration, i.e., at event times.

We haven't talked yet about the gate opening event. We cannot handle the gate opening event in the same fashion as the gate closing event, because we don't know beforehand, *when* the gate will open. We only know, *under what condition* this will be the case, namely when the current that flows through the thyristor becomes negative.

The gate opening event will be discussed in due course.

9.4 Simulation of Sampled–data Systems

A typical application of time events is the simulation of sampled–data control systems. A continuous–time plant is being controlled by one or several discrete–time controllers that may operate on the same or on different frequencies (multi–rate sampling).

A typical application is shown in Fig.9.11.

A robot arm is to be controlled by one or several computers. The innermost control loop serves the purposes of stabilization, linearization, and

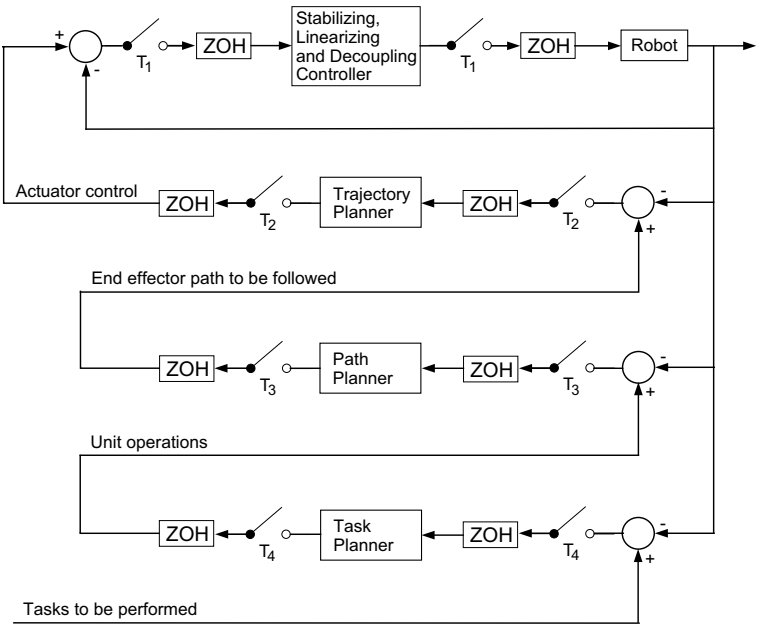

FIGURE 9.11. Robot control.

decoupling. The purpose of this controller is to make the larger control issues easier to tackle. The signal needs to be sampled in short time intervals, T_1, in order to keep the control loop stable. This first controller is then added to the plant, i.e., the next higher–level controller considers the innermost control loop part of the plant to be controlled. Its purpose is to translate a desired path into control signals for the actuators of the motors that drive the individual joints of the robot arm. This controller solves the dynamic control problem. It can operate at a slightly slower sampling rate, T_2, than the stabilizing controller. The next higher–level controller solves the static control problem. It translates descriptions of individual unit operations into desired end–effector positions expressed as functions of time. It again can operate at a somewhat reduced sampling rate, T_3. Finally, the task planner decomposes complex tasks into series of unit operations that it then submits to the path planner for execution. The task planner can operate at a considerably slower sampling rate, T_4. Thus:

$$T_1 \leq T_2 \leq T_3 \ll T_4 \tag{9.6}$$

Figure 9.11 is somewhat stylized. There are multiple signals to be fed back, and after decoupling, there may be multiple control loops, one for each joint.

The simulation program will contain a single dynamic block describing the motion of the robot arm itself together with its motors and drive trains

in between events. The program also contains four separate discrete blocks, one for each controller, that are executed at different, yet previously known, points in time. All four discrete controllers are probably scheduled to be executed for the first time during initialization of the simulation. Thus, we are confronted with four simultaneous events, and it will be important that the task planner is executed first, then the path planner, then the trajectory planner, and finally the stabilizer, since the inner control loops need the set points from the outer control loops to function properly. Each controller will, as part of its event description, schedule the next execution of itself to occur T_i time units into the future. At a later time, it is probably better to resolve ties by assigning a higher priority to the inner control loops, since they are more time–critical.

At any point in time, there are thus scheduled four different time events to take place at different time instants in the future. These are maintained by the so–called *event queue*, which is usually implemented as a linear linked list with pointers back and forth, in which future events are placed in ascending order of execution time using additional rules for tie breaking.

9.5 State Events

Frequently, the time of occurrence of a discontinuity is not known in advance. For example in the thyristor circuit, it is not known in advance when the thyristor will open again. All we know is that it will open when the current passes through zero. Thus, we know the *event condition*, rather than the *event time*, specified in terms of a function of continuously varying simulation variables.

Event conditions are usually specified implicitly, i.e., in the form of *zero–crossing functions*. A state event occurs when a variable associated with it crosses through zero. Multiple zero–crossing functions may be associated with a single event type.

The zero–crossing functions must be tested continuously during simulation. Thus, they are part of the continuous system simulation environment. To this end, many of the numerical ODE solvers currently on the market offer so–called *root solvers*. Variables to be tested for zero crossing are placed in a vector. These variables are monitored constantly during simulation, and if one of them passes through zero, an iteration is started to determine the zero–crossing time with a pre–specified precision.

Since we don't know when event conditions become true, we cannot reduce the step size to hit them accurately. Instead, we need some sort of iteration (or interpolation) mechanism to locate the event time. Thus, when an event condition is alerted during the execution of an integration step, it influences the step–size control mechanism of the integration algorithm by forcing the continuous simulation to iterate (or interpolate) to the earliest

zero–crossing within the current integration step.

9.5.1 Multiple Zero Crossings

Figure 9.12 illustrates the iteration of event conditions, assuming that multiple zero crossings have taken place within a single integration step.

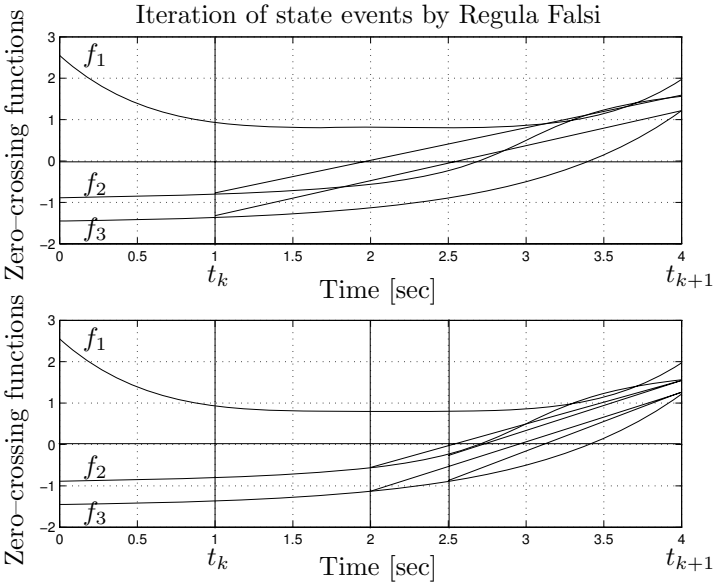

FIGURE 9.12. Iteration of multiple event conditions using Regula Falsi.

Figure 9.12 shows three different zero–crossing functions, f_1, f_2, and f_3. At time $t_k = 1.0$, f_1 is positive, whereas f_2 and f_3 are negative. We perform an integration step of length $h = 3.0$. At time $t_{k+1} = 4.0$, f_1 is still positive, whereas both f_2 and f_3 are now also positive, i.e., two zero crossings have taken place within this integration step.

We connect the end points of each zero–crossing function, determine, where these straight lines cross through zero, and choose the smallest of these time instants as the next time point. Mathematically:

$$t_{\text{next}} = \min_{\forall i} \left[\frac{f_i(t_{k+1}) \cdot t_k - f_i(t_k) \cdot t_{k+1}}{f_i(t_{k+1}) - f_i(t_k)} \right] \tag{9.7}$$

where i stretches over all functions with a zero crossing within the interval. Thus, we repeat the last time step with a step size of $h = t_{\text{next}} - t_k$.

If no zero crossing has taken place during the reduced step from t_k to t_{next}, we accept t_{next} as t_k and repeat the algorithm using the remainder of the interval.

If more than one zero crossing has taken place during the reduced interval, we reduce t_{k+1} to t_{next}, and apply the same algorithm once more to the so reduced interval.

If exactly one zero crossing has taken place during the reduced interval, we have simplified the problem to that of finding the zero crossing of a single zero–crossing function, for which a number of algorithms can be used that shall be presented in due course.

The algorithm converges always, as the interval is reduced during each iteration step. Unfortunately, it is not possible to estimate the number of iteration steps needed until convergence has been reached using this method. Convergence can indeed be quite slow.

Another algorithm that is sometimes used instead is the *Golden Section* method. The Golden Section method has the advantage that, in each iteration step, the interval is reduced by a fixed ratio. Thus, the interval will soon become quite small. This is how it works.

Already the ancient Greeks had discovered that there exists a special rectangle with the property that if one cuts off a square, the remaining rectangle has the same proportions as the original one. This is shown in Fig.9.13.

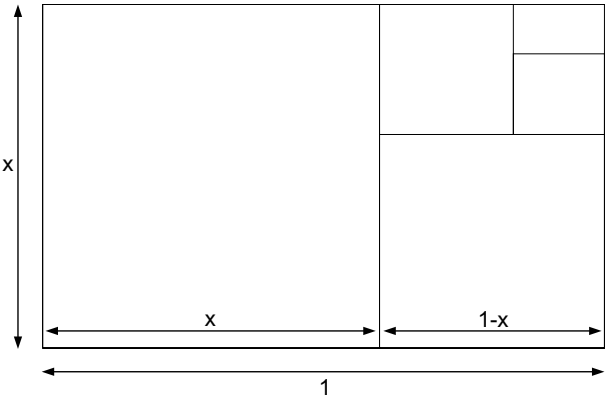

FIGURE 9.13. The Golden Section.

Thus:

$$\frac{x}{1} = \frac{1-x}{x} \tag{9.8}$$

which leads to $x = 0.618$.

This idea can be applied to the problem of isolating individual zero–crossing functions. The method is shown in Fig.9.14.

Once the iteration algorithm has been triggered by multiple zero crossings within a single step, the interval is subdivided by calculating two partial steps, one of length $(1-x)\cdot h$, the other of length $x \cdot h$. Both of these partial steps start at time t_k. In this way, the interval is subdivided

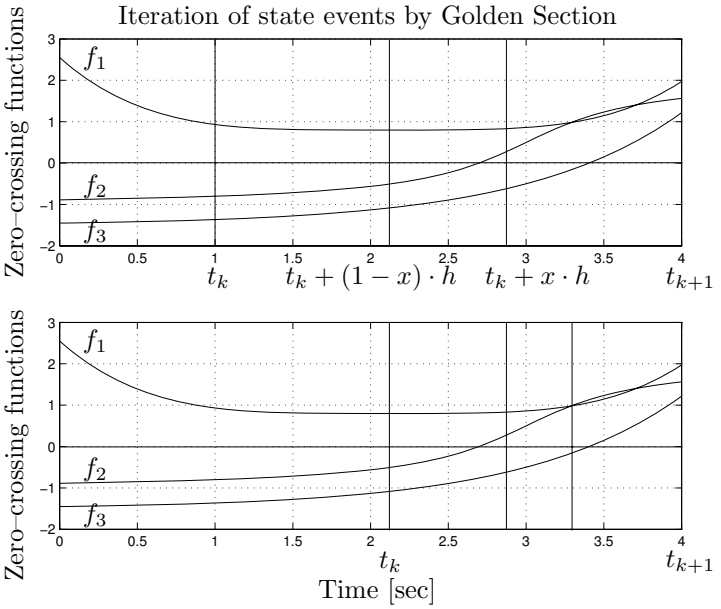

FIGURE 9.14. Iteration of multiple event conditions using Golden Section.

into three subintervals.

If there is no zero crossing within the leftmost of these three subintervals, then that subinterval can be thrown away, i.e. t_k is updated to $t_k+(1-x)\cdot h$, and a new partial step is computed, as shown in the lower part of Fig.9.14.

If there are multiple zero crossings within the leftmost of the three subintervals, then the rightmost subinterval is discarded, t_{k+1} is updated to $t_k + x \cdot h$, and a new partial step is computed, always keeping the proportions of the three subintervals the same.

If there is exactly one zero crossing within the leftmost of the four subintervals, then t_{k+1} is updated to $t_k+(1-x)\cdot h$, and we continue with any one of the algorithms for finding a single zero crossing within a given interval.

The Golden Section algorithm can be slightly improved using a *Fibonacci Series* instead, but this is hardly ever worth it. The Fibonacci Series shrinks the interval slightly faster than the Golden Section technique, but it can be shown that the Fibonacci Series is always less than one iteration step ahead of Golden Section, and it is only better at all, if we decide up front how many iteration steps we are going to perform altogether.

9.5.2 Single Zero Crossings, Single–step Algorithms

Of course, any of the techniques presented so far for isolating individual zero–crossing functions can also be used to find the zero crossings themselves. Yet, this may be inefficient, as all of these techniques offer only

linear convergence speed.

All simulation variables in a state–space model can ultimately be expressed in terms of state variables and inputs only. This also applies to the zero–crossing functions. Thus, we could use, in the determination of the zero crossings, not only the values of the zero–crossing functions themselves at different points in time, but also the values of their derivatives.

A first algorithm that exploits this possibility is the well-known *Newton iteration* algorithm that we have used so often already in this book, albeit for different purposes. Figure 9.15 documents the approach.

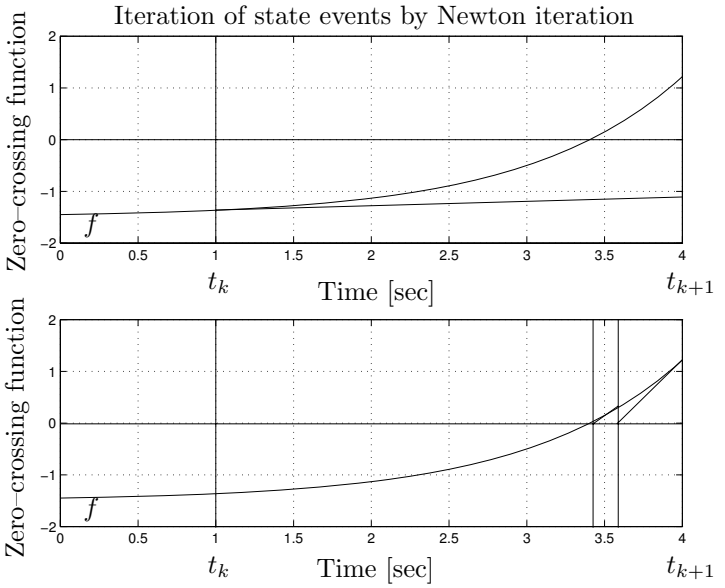

FIGURE 9.15. Iteration on single zero–crossing functions using Newton iteration.

Once a zero–crossing function has been isolated, we can use either t_k or t_{k+1} as the starting point of a Newton iteration.

The good news about Newton iteration is that the algorithm exhibits a *quadratic convergence speed.* Thus, Newton iteration converges much more rapidly than either Regula Falsi or Golden Section, if the algorithm converges at all.

Unfortunately, and contrary to the previously introduced algorithms, the Newton iteration algorithm does not always converge. In the given example, if we start at t_{k+1}, the algorithm converges quickly, whereas if we start at t_k, already the next step takes the algorithm far outside the interval $[t_k, t_{k+1}]$.

Furthermore, it may not be easy to determine upfront, whether or not the algorithm will converge on a given example. For these reasons, Newton iteration may not be the method of choice to be used as a root solver.

A better approach may be to use the derivative values at both ends of

the interval $[t_k, t_{k+1}]$ simultaneously. Since we have access to four pieces of information: f_k, df_k/dt, f_{k+1}, and df_{k+1}/dt, we can lay a third-order polynomial through these four pieces of information and solve for its roots.

The interpolation polynomial can thus be written as:

$$p(t) = a \cdot t^3 + b \cdot t^2 + c \cdot t + d \tag{9.9}$$

with the derivative:

$$\dot{p}(t) = 3a \cdot t^2 + 2b \cdot t + c \tag{9.10}$$

Thus, we can write the four pieces of information as follows:

$$p(t_k) = a \cdot t_k^3 + b \cdot t_k^2 + c \cdot t_k + d = f_k \tag{9.11a}$$

$$p(t_{k+1}) = a \cdot t_{k+1}^3 + b \cdot t_{k+1}^2 + c \cdot t_{k+1} + d = f_{k+1} \tag{9.11b}$$

$$\dot{p}(t_k) = 3a \cdot t_k^2 + 2b \cdot t_k + c = \dot{f}_k = h_k \tag{9.11c}$$

$$\dot{p}(t_{k+1}) = 3a \cdot t_{k+1}^2 + 2b \cdot t_{k+1} + c = \dot{f}_{k+1} = h_{k+1} \tag{9.11d}$$

which can be written in matrix/vector form as:

$$\begin{pmatrix} f_k \\ f_{k+1} \\ h_k \\ h_{k+1} \end{pmatrix} = \begin{pmatrix} t_k^3 & t_k^2 & t_k & 1 \\ t_{k+1}^3 & t_{k+1}^2 & t_{k+1} & 1 \\ 3t_k^2 & 2t_k & 1 & 0 \\ 3t_{k+1}^2 & 2t_{k+1} & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} \tag{9.12}$$

Equation 9.12 can then be solved for the unknown coefficients a, b, c, and d.

Figure 9.16 illustrates the method.

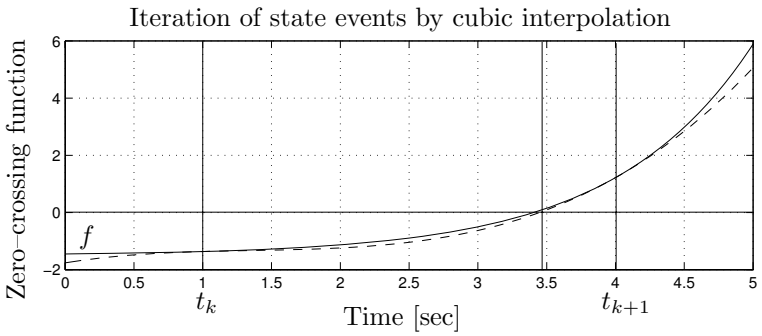

FIGURE 9.16. Iteration on single zero–crossing functions using cubic interpolation.

The method converges even faster than Newton iteration, as it exhibits *cubic convergence speed*. Furthermore, it is guaranteed to converge, just like Regula Falsi and Golden Section.

The cubic polynomial must have at least one real solution within the interval $[t_k, t_{k+1}]$. Possibly there are three real solutions within the interval, in which case any one of them could be used as the next evaluation time, t_{next}.

Yet, we may be able to improve on that method even a little further. One drawback of the proposed technique is that we need to solve for the roots of a cubic polynomial to determine a real root that lies inside the interval $[t_k, t_{k+1}]$.

Instead of fitting a cubic polynomial as proposed above, we could also fit an *inverse cubic polynomial* of the type:

$$t(p) = a_1 \cdot p^3 + b_1 \cdot p^2 + c_1 \cdot p + d_1 \tag{9.13}$$

which can simply be evaluated for $p = 0$. Thus, the next evaluation time can be computed as:

$$t_{\text{next}} = t(p = 0) = d_1 \tag{9.14}$$

The following four pieces of information are at our disposal:

$$t_k = t(f_k) \tag{9.15a}$$
$$t_{k+1} = t(f_{k+1}) \tag{9.15b}$$
$$u_k = \frac{dt(f_k)}{df} = \frac{1}{h_k} \tag{9.15c}$$
$$u_{k+1} = \frac{dt(f_{k+1})}{df} = \frac{1}{h_{k+1}} \tag{9.15d}$$

We know that:

$$t_k = a_1 \cdot f_k^3 + b_1 \cdot f_k^2 + c_1 \cdot f_k + d_1 \tag{9.16a}$$
$$t_{k+1} = a_1 \cdot f_{k+1}^3 + b_1 \cdot f_{k+1}^2 + c_1 \cdot f_{k+1} + d_1 \tag{9.16b}$$
$$u_k = 3a_1 \cdot f_k^2 + 2b_1 \cdot f_k + c_1 \tag{9.16c}$$
$$u_{k+1} = 3a_1 \cdot f_{k+1}^2 + 2b_1 \cdot f_{k+1} + c_1 \tag{9.16d}$$

or in matrix form:

$$\begin{pmatrix} t_k \\ t_{k+1} \\ u_k \\ u_{k+1} \end{pmatrix} = \begin{pmatrix} f_k^3 & f_k^2 & f_k & 1 \\ f_{k+1}^3 & f_{k+1}^2 & f_{k+1} & 1 \\ 3f_k^2 & 2f_k & 1 & 0 \\ 3f_{k+1}^2 & 2f_{k+1} & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \end{pmatrix} \tag{9.17}$$

which could be solved directly for the four unknowns by means of Gaussian elimination.

Yet, we can do even better. We shall use *inverse Hermite interpolation*. The scheme is called inverse interpolation, since we fit the inverse function with the polynomial. The polynomials that we shall use to span our base are Hermite polynomials.

We shall introduce a new variable ϕ of the type:

$$\phi = coef_1 \cdot f + coef_2 \tag{9.18}$$

such that:

t	f	ϕ
t_k	f_k	0.0
t_{k+1}	f_{k+1}	1.0
t_{next}	0.0	$\hat{\phi}$

TABLE 9.1. Variable transformation.

We find that:

$$coef_1 = \frac{1}{f_{k+1} - f_k} \tag{9.19a}$$

$$coef_2 = -\frac{f_k}{f_{k+1} - f_k} = \hat{\phi} \tag{9.19b}$$

We now construct four auxiliary polynomials in ϕ:

$$p_i(\phi) = \alpha_i \cdot \phi^3 + \beta_i \cdot \phi^2 + \gamma_i \cdot \phi + \delta_i \tag{9.20a}$$

$$\frac{dp_i(\phi)}{d\phi} = 3\alpha_i \cdot \phi^2 + 2\beta_i \cdot \phi + \gamma_i \tag{9.20b}$$

such that:

$$p_1(0) = 1 \quad ; \quad p_1(1) = 0 \quad ; \quad \frac{dp_1(0)}{d\phi} = 0 \quad ; \quad \frac{dp_1(1)}{d\phi} = 0 \tag{9.21a}$$

$$p_2(0) = 0 \quad ; \quad p_2(1) = 1 \quad ; \quad \frac{dp_2(0)}{d\phi} = 0 \quad ; \quad \frac{dp_2(1)}{d\phi} = 0 \tag{9.21b}$$

$$p_3(0) = 0 \quad ; \quad p_3(1) = 0 \quad ; \quad \frac{dp_3(0)}{d\phi} = 1 \quad ; \quad \frac{dp_3(1)}{d\phi} = 0 \tag{9.21c}$$

$$p_4(0) = 0 \quad ; \quad p_4(1) = 0 \quad ; \quad \frac{dp_4(0)}{d\phi} = 0 \quad ; \quad \frac{dp_4(1)}{d\phi} = 1 \tag{9.21d}$$

It is easy to verify that these polynomials are:

$$p_1(\phi) = 2\phi^3 - 3\phi^2 + 1 \tag{9.22a}$$

$$p_2(\phi) = -2\phi^3 + 3\phi^2 \tag{9.22b}$$

$$p_3(\phi) = \phi^3 - 2\phi^2 + \phi \tag{9.22c}$$

$$p_4(\phi) = \phi^3 - \phi^2 \tag{9.22d}$$

The inverse Hermite interpolation polynomial:

$$p(\phi) = a_2 \cdot \phi^3 + b_2 \cdot \phi^2 + c_2 \cdot \phi + d_2 \tag{9.23}$$

now expressed as a function of ϕ rather than of f, can be written in these auxiliary polynomials as:

$$p(\phi) = t_k \cdot p_1(\phi) + t_{k+1} \cdot p_2(\phi) + s_k \cdot p_3(\phi) + s_{k+1} \cdot p_4(\phi) \tag{9.24}$$

where:

$$s_k = \frac{dt_k}{d\phi} = \frac{1}{d\phi_k/dt} = \frac{1}{coef_1 \cdot (df_k/dt)} = \frac{f_{k+1} - f_k}{h_k} \tag{9.25a}$$

$$s_{k+1} = \frac{f_{k+1} - f_k}{h_{k+1}} \tag{9.25b}$$

In order to obtain the desired zero–crossing time, t_{next}, we simply evaluate Eq.(9.24) at $\phi = \hat{\phi}$.

Figure 9.17 illustrates the method.

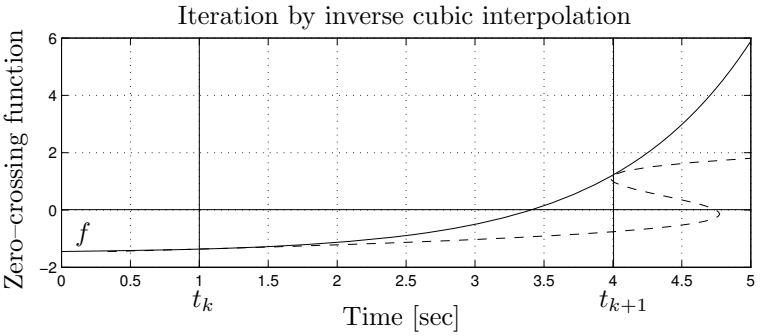

FIGURE 9.17. Iteration on single zero–crossing functions using inverse cubic interpolation.

Inverse Hermite interpolation is certainly more elegant than direct cubic interpolation. Unfortunately, the simplification in the computation came at a dire price, as we lost our guaranteed convergence. We can no longer guarantee that the solution is to be found within the interval $[t_k, t_{k+1}]$, and in the given example, this indeed is not the case.

Notice that all of these techniques were used only to determine the next time instant, t_{next}, for evaluating the zero–crossing function. The actual computation of the zero–crossing function is done by means of numerical integration, i.e., using the same higher–order numerical integration scheme used throughout the simulation. Thus, no approximation accuracy is lost in the process.

9.5.3 Single Zero Crossings, Multi–step Algorithms

In the case of multi–step algorithms, we may be able to do even better [9.3]. At the end of the step that puts the event conditions on alert, i.e., at

time t_{k+1}, we have the Nordsieck vector available. Thus, we can write:

$$\mathcal{F}(\hat{h}) = \mathcal{F}_i(t_{\text{next}}) = \mathcal{F}_i(t_{k+1}) + \hat{h}\frac{d\mathcal{F}_i(t_{k+1})}{dt} + \frac{\hat{h}^2}{2}\frac{d^2\mathcal{F}_i(t_{k+1})}{dt^2} + \frac{\hat{h}^3}{6}\frac{d^3\mathcal{F}_i(t_{k+1})}{dt^3} + \cdots = 0.0 \tag{9.26}$$

This is a function in the unknown $\hat{h}$ that can be solved by Newton iteration. We set:

$$\hat{h}^0 = 0.5 \cdot (t_k - t_{k+1}) \tag{9.27}$$

and iterate:

$$\hat{h}^{\ell+1} = \hat{h}^\ell - \frac{\mathcal{F}(\hat{h}^\ell)}{\mathcal{H}(\hat{h}^\ell)} \tag{9.28}$$

where:

$$\mathcal{H}(\hat{h}) = \frac{d\mathcal{F}(\hat{h})}{d\hat{h}} = \frac{d\mathcal{F}_i(t_{k+1})}{dt} + \hat{h}\frac{d^2\mathcal{F}_i(t_{k+1})}{dt^2} + \frac{\hat{h}^2}{2}\frac{d^3\mathcal{F}_i(t_{k+1})}{dt^3} + \dots \tag{9.29}$$

Using this technique, we can determine the time of the zero–crossing in a single step with the same accuracy as the integration itself. However, we have the Nordsieck vector only available for *state variables*, not for *algebraic variables*. Therefore, it is useful to treat event conditions as additional state variables, by writing:

$$x_{n+i} = \mathcal{F}_i(\mathbf{x}) \tag{9.30a}$$

$$\dot{x}_{n+i} = \frac{d\mathcal{F}_i(\mathbf{x})}{dt} \tag{9.30b}$$

For the benefit of improved accuracy, it is probably a good idea to keep both equations in the model rather than integrating Eq.(9.30b) into Eq.(9.30a). However, the variables will be treated like additional state variables, and will be maintained by the integration algorithm in its data base of old values. In this way, it is possible to compute the Nordsieck vector for event conditions whenever needed.

We shall need to compute Eq.(9.30b) anyway, since otherwise, we cannot conveniently apply an iteration procedure other than Regula Falsi or Golden Section.

9.5.4 Non–essential State Events

Sometimes, it may be a good idea to even add Eq.(9.30b) as a *non–essential event condition* to the set of event conditions. Figure 9.18 illustrates the reason for this suggestion.

f_1 is an essential event condition, whereas $f_2 = \dot{f}_1$ is a non–essential event condition. A non–essential event condition is an event condition that doesn't have an event action associated with it.

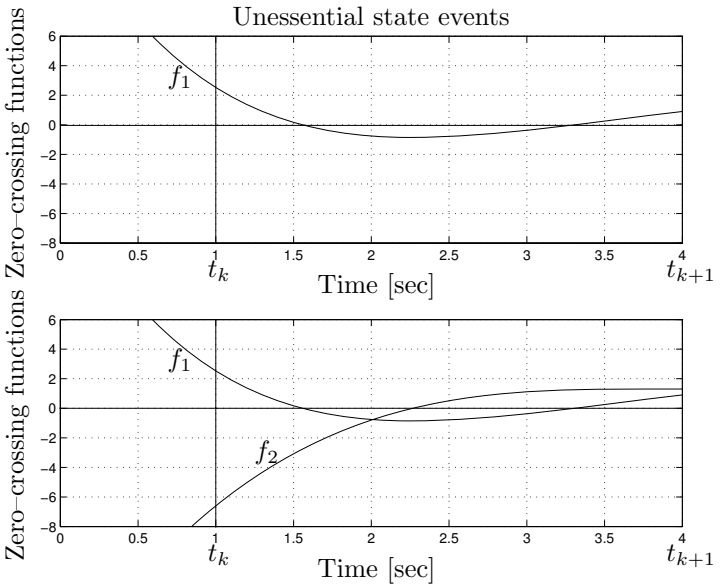

FIGURE 9.18. Non–essential event conditions.

Had we only formulated f_1 as a zero–crossing function, the event at time $t = 1.55$ would have been missed, because the essential zero–crossing function, f_1, crosses through zero twice within the single integration step from time $t_k = 1.0$ to time $t_{k+1} = 4.0$.

Adding the non–essential zero–crossing function, f_2, to the set of zero–crossing functions solves the dilemma, because f_2 exhibits a zero crossing, whenever f_1 goes through an extremum.

During the iteration of the non–essential event condition, f_2, the algorithm will discover that also f_1 crosses through zero, and will iterate on that zero crossing first, as it happens earlier.

9.6 Consistent Initial Conditions

Figure 9.19 shows a piecewise linear function with three segments. In the "left" region, $y = a_1 \cdot x + b_1$, in the "center" region, $y = a_2 \cdot x + b_2$, and in the "right" region, $y = a_3 \cdot x + b_3$.

Traditionally, we would describe such a function using an if–statement:

if $x < x_1$ **then** $y = a_1 \cdot x + b_1$
else if $x < x_2$ **then** $y = a_2 \cdot x + b_2$
else $y = a_3 \cdot x + b_3$;

However, we know meanwhile that, if the variable y is used in a state–space model, this will force the step–size control mechanism to reduce the step

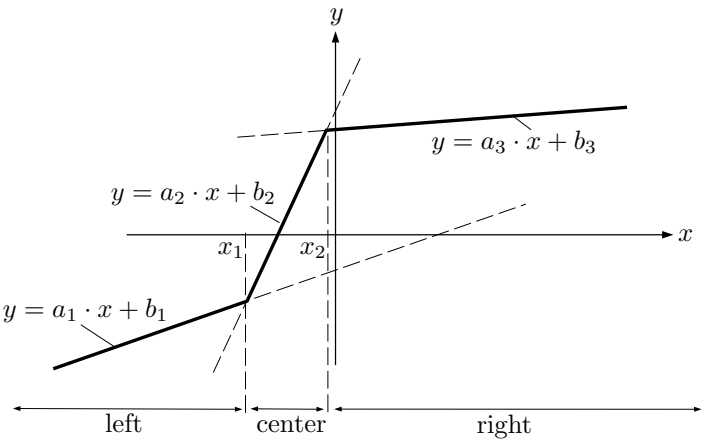

FIGURE 9.19. Discontinuous function.

size, whenever x crosses through one of the two thresholds, x_1 or x_2, within an integration step.

Thus, we may choose to program the function using three different models, one for each region, with appropriate zero–crossing functions describing the conditions for switching from one region to the next.

In pseudo–code, we might write:

```
case region
   left :    y = a1 · x + b1;
             schedule Center when x − x1 == 0;
   center :  y = a2 · x + b2;
             schedule Left when x − x1 == 0;
             schedule Right when x − x2 == 0;
   right :   y = a3 · x + b3;
             schedule Center when x − x2 == 0;
end;
```

together with the three discrete event descriptions:

```
event Left
   region := left;
end Left;

event Center
   region := center;
end Center;

event Right
   region := right;
end Right;
```

The *schedule*–statements are used in this pseudo–code to describe zero–crossing functions. The variable *region* is not a continuously changing vari-

able. From the point of view of the continuous simulation, it assumes the role of a parameter. Its value can only change within a discrete event description.

This should work except for one little detail. The variable *region* is a *discrete state variable* that needs to be initialized. Somewhere in the initial region of the simulation program, we would need a statement such as:

if $x < x_1$ **then** $region := left$;
else if $x < x_2$ **then** $region := center$;
else $region := right$;

Will this code always work? Unfortunately, the answer to that question is no. One problem that we haven't considered yet is that x may reach one of the thresholds without actually crossing through it.

Let us assume that:

$$x(t) = \frac{x_2 - x_1}{2} \cdot \sin(t) + \frac{x_1 + x_2}{2} \tag{9.31}$$

In this case, x will always remain in the center region. It will only just reach the two thresholds, x_1 and x_2, every once in a while.

The event description, as programmed above, would make the model switch regions, each time a threshold is reached. One of the more difficult problems associated with the simulation of discontinuous functions is to know, in which region the model operates after the event has been processed, i.e., to find a consistent set of initial conditions after event handling.

The problem is by no means an academic one. Consider the case of a set of bowling balls resting on a guide rail. They are all in contact with each other. A new ball arrives with velocity v that hits the first of these balls. We all know what will happen: the new ball will come to rest at once, and the last of the previously resting balls will move away with the same speed v. Yet, convincing a simulation program that this is what must happen is anything but trivial.

One way to deal with this problem is to define a narrow band around each of the zero–crossing functions. The event is detected when the function crosses through zero, at which time the event is being processed. Yet, before starting with the next continuous–time simulation segment, trial steps are taken to determine whether or not the zero-crossing functions will leave the bands placed around the zero crossing as expected. It happens frequently that one event immediately triggers other events that change the condition on the original event again.

An example of this problem might be a robot arm with sticking friction in each of its articulations [9.11]. Once the force in an articulation overcomes sticking friction, the articulation starts to move. Yet, this immediately changes the forces in neighboring articulations. As a consequence, another neighboring articulation may come out of sticking friction also, which changes the forces in the articulations once again, with the possible

effect that the original articulation returns back to its sticking region.

Modeling this situation correctly is anything but trivial. Let us attempt this task. Figure 9.20 shows a typical friction model with sticking friction, dry (Coulomb) friction, and viscous friction.

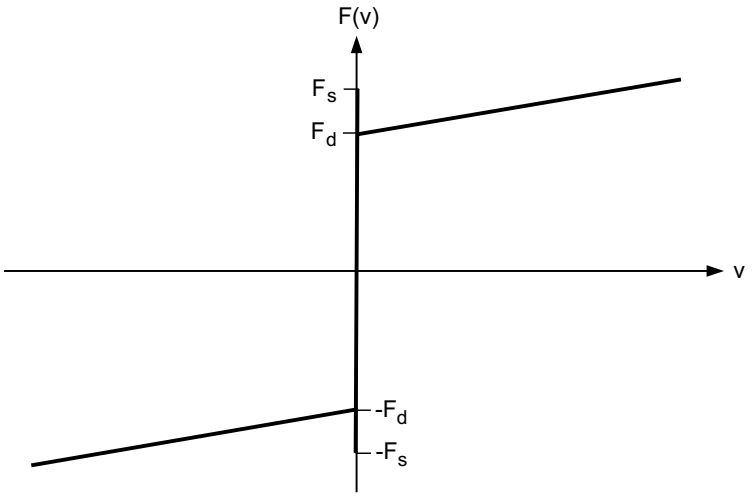

FIGURE 9.20. Friction characteristic.

There are three different regions (modes) of this nonlinear model: a *backward* mode, a *sticking* mode, and a *forward* mode. While the velocity of the articulation is zero, the articulation operates in its sticking mode. It will remain in this mode, until the sum of forces applied to this articulation becomes either larger than the positive sticking friction force, F_s, or smaller than the negative sticking friction force, $-F_s$. When this happens, the articulation comes out of sticking friction, and changes its operational mode to moving either *forward* or *backward*, in which the friction force is computed as the sum of a dry (Coulomb) friction component, $\pm F_d$, and a linear (viscous) friction component. Once the model operates in one of its two moving modes, it will remain in that mode, until the velocity of the articulation crosses through zero, at which time the model will return to its sticking mode.

Yet, this model is still too simple, as it does not account for the possibility that the result of coming out of sticking friction might be to return to sticking friction immediately again, after having freed up another articulation in the process.

A more complete model is shown in Figure 9.21, which exhibits a *state transition diagram* of the friction characteristic.

The new model possesses six different modes. Beside from the three modes used in the earlier model, we also have a *Start Forward* mode, and a *Start Backward* mode. If the sum of all forces applied to the articulation

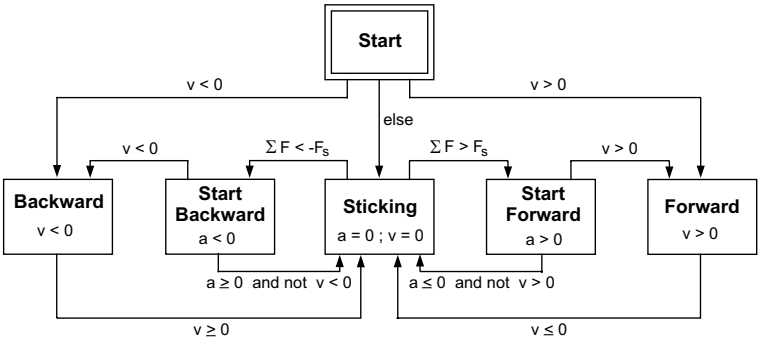

FIGURE 9.21. State transition diagram of friction characteristic.

is larger than zero, the articulation leaves the sticking mode. However, it doesn't proceed immediately to the *Forward* mode yet. Instead, it enters a transitory mode, called the *Start Forward* mode. As the sum of all forces is larger than zero, also the acceleration, in accordance with *Newton's law*, is now positive. However, the velocity is initially still zero. It will only become positive through integration, as time proceeds forward.

However, before integration starts again, the new forces are propagated to the neighboring bodies, possibly taking some other articulations also out of their sticking mode. As the condition for mode switching is programmed in the form of *state conditions*, rather than *time events*, integration needs to start, in order for this propagation to take place.

As integration starts, multiple zero–crossing functions may be triggered during the first new integration step. One would be to take the original articulation from the *Start Forward* mode to the *Forward* mode. Another may be to take a neighboring articulation from the *Sticking* mode to the *Start Forward* mode.

The model must make sure that the latter event takes precedence over the former. This is accomplished by recognizing the velocity as being positive only, after the velocity has become larger than some fudge factor $v > \varepsilon$, which implements the narrow band around the zero crossings that we wrote about earlier.

The *Start* mode implements the initialization of the discrete state variable. The discrete state variable starts out in its *Start* mode, from where it proceeds immediately to one of the other modes, depending on the initial velocity.

9.7 Object–oriented Descriptions of Discontinuities

What a mess have we created here! In order to protect the integration algorithm from having to integrate across discontinuities, we introduced

two new modeling elements: *time events* and *state events*, that make the simulation of discontinuous models safer and faster, but make the modeling task quite a bit more complicated.

Is this impressively complicated apparatus really necessary? The answer to this question is yes and no. On the one hand, we truly require integration algorithms with root solvers for safe and efficient discontinuity handling. We also require time events for the description of discontinuities that take place at previously known event times. Yet, state event descriptions are not a sufficiently high–level mechanism to bother the average simulation practitioner with.

Although the simulation code, i.e., the code used by the numerical integration algorithm, may have to be complex and messy, this doesn't mean that the modeler has to manually enter it in this fashion.

Returning once more to the example of Fig. 9.19. What is wrong with a description of the type:

$$
\begin{aligned}
y = \; & \textbf{if } x < x_1 \textbf{ then } a_1 \cdot x + b_1 \\
& \textbf{else if } x < x_2 \textbf{ then } a_2 \cdot x + b_2 \\
& \textbf{else } a_3 \cdot x + b_3;
\end{aligned}
$$

to describe what this function does? It expresses perfectly well and in an unambiguous fashion, what the model is supposed to do. Can't we build a *model compiler* that takes such a description, and translates it down to the level of state events at compile time?

This is the approach that was taken in the design of the Dymola modeling environment [9.11, 9.12], and indeed, the syntax of the program segment shown above is that of Dymola.

Already in the previous two chapters, we encountered the need for symbolic preprocessing of model equations, in order to obtain numerically suitable simulation code. Although we applied these symbolic graph coloring algorithms in a manual fashion, by manually causalizing the structure digraph, this can obviously only be done for toy problems, such as the simple electrical circuits used to introduce the algorithms.

In a realistically complex model, such as a six–degree–of–freedom robot arm, leading to possibly 10,000 equations initially, it must be possible to apply all of these algorithms in a completely automated fashion. This is what the Dymola model compiler does [9.4]. The algorithms implemented in Dymola [9.12] are essentially those that were introduced in the previous two chapters.

Yet, Dymola is capable of performing considerably more complex model compilations, as it decomposes object–oriented descriptions of discontinuous models into suitable event descriptions at compile time.

Up to this point, we were able to either describe our algorithms in MATLAB, or apply them manually, as we did with some of the algorithms in the previous two chapters. Now, we don't have that luxury any longer, as

even simple functions, such as the friction model introduced earlier, quickly become too involved to conveniently describe them as a collection of event descriptions.

Thus, we shall need to introduce some of the low–level modeling constructs of Dymola [9.12] at this time to be able to describe the necessary discontinuity handling algorithms in a suitably compact fashion.

9.7.1 The Computational Causality of if–Statements

We have seen in the previous two chapters that the computational causality of statements should not be predetermined, but must be allowed to vary depending on the embedding of the objects containing these statements within their environment.

The equal sign of an equation is not to be interpreted as an *assignment* in the usual sense of sequential programming languages, but rather as an *equality* in the algebraic sense.

Hence in a Dymola program, it doesn't matter whether Ohm's law is formulated as:

$$u = R * i$$

or:

$$i = u/R$$

or finally:

$$0 = u - R * i$$

Dymola will treat each of these statements in exactly the same fashion. It will turn equations around symbolically as needed.

It may now have become clear, why the Dymola syntax for the *if*–statement of the nonlinear characteristic of Fig. 9.19 is:

$y =$ **if** $x < x_1$ **then** $a_1 \cdot x + b_1$
else if $x < x_2$ **then** $a_2 \cdot x + b_2$
else $a_3 \cdot x + b_3$;

rather than:

if $x < x_1$ **then** $y = a_1 \cdot x + b_1$
else if $x < x_2$ **then** $y = a_2 \cdot x + b_2$
else $y = a_3 \cdot x + b_3$;

Dymola needs to ensure that each branch of the *if*–statement computes the same variable, as otherwise, the *vertical sorting* algorithm would invariably fail.

Do *if*–statements have a fixed computational causality, or is it possible to turn them around in the same way as we turn around algebraic equations?

To answer this question, let us translate the above *if*–statements to an event description that looks a bit different from the one used before. To this end, we shall introduce three additional integer variables, m_l, m_c, and m_r, whose values are linked to the linguistic discrete state variable, *region*, in the following way:

region	m_l	m_c	m_r
left	1	0	0
center	0	1	0
right	0	0	1

Using these new variables, the event description of the nonlinear characteristic can be rewritten as follows:

$y = m_l \cdot (a_1 \cdot x + b_1) + m_c \cdot (a_2 \cdot x + b_2) + m_r \cdot (a_3 \cdot x + b_3);$
case *region*
  left : **schedule** *Center* **when** $x - x_1 == 0;$
  center : **schedule** *Left* **when** $x - x_1 == 0;$
  schedule *Right* **when** $x - x_2 == 0;$
  right : **schedule** *Center* **when** $x - x_2 == 0;$
end;

together with the three discrete event descriptions:

event *Left*
  region := *left*;
  $m_l = 1; \quad m_c = 0; \quad m_r = 0;$
end *Left*;

event *Center*
  region := *center*;
  $m_l = 0; \quad m_c = 1; \quad m_r = 0;$
end *Center*;

event *Right*
  region := *right*;
  $m_l = 0; \quad m_c = 0; \quad m_r = 1;$
end *Right*;

In this way, the former *if*–statement has been converted to the algebraic statement:

$$y = m_l \cdot (a_1 \cdot x + b_1) + m_c \cdot (a_2 \cdot x + b_2) + m_r \cdot (a_3 \cdot x + b_3) \qquad (9.32)$$

which can be turned around in the usual way:

$$x = \frac{y - m_l \cdot b_1 - m_c \cdot b_2 - m_r \cdot b_3}{m_l \cdot a_1 - m_c \cdot a_2 - m_r \cdot a_3} \tag{9.33}$$

as long as none of the three slopes is flat, i.e., as long as none of the parameters a_1, a_2, or a_3 is equal to zero.

9.7.2 Multi–valued Functions

The *if*–statements that we have introduced so far don't allow the description of multi–valued functions, such as the dry hysteresis function of Fig. 9.22.

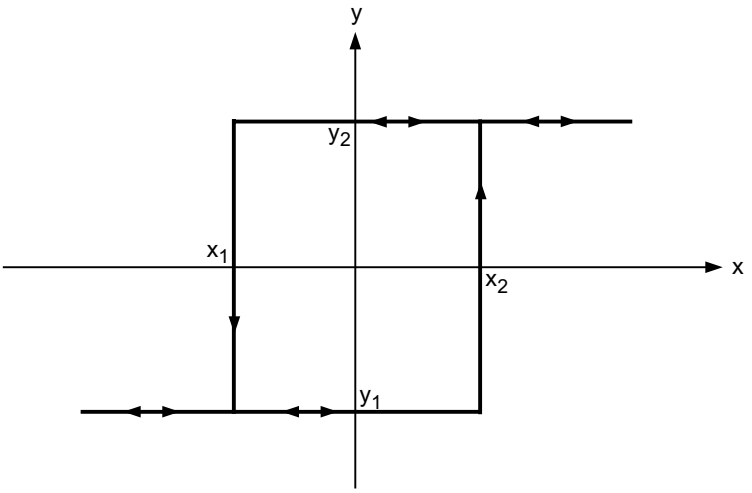

FIGURE 9.22. Dry hysteresis function.

A possible event description for the dry hysteresis function could look as follows:

```
y = y_last;
case region
   up :     schedule Down when x − x1 == 0;
   down :   schedule Up when x − x2 == 0;
end;
```

together with the two discrete event descriptions:

```
event Up
   region := up;
   y_last := y2;
end Left;
```

```
event Down
   region := down;
   y_last := y1;
end Center;
```

Dymola offers a *when*–statement that allows to encode such an event description in a compact form. We could try to encode the dry hysteresis function as follows:

```
when x < x1
   y = y1;
end when;

when x > x2
   y = y2;
end when;
```

Contrary to the *if*–statement that takes the semantics of "if is," the *when*–statement has associated with it the semantics "when becomes." Thus, the former of the two *when* clauses will only be executed, whenever x becomes smaller than x_1, whereas the latter of the two *when* clauses will only be executed, whenever x becomes larger than x_2. At all other times, y simply retains its former value.

Consequently, we shall require an appropriate initialization section to provide an initial value for the discrete state variable, y.

Unfortunately, the above program won't work correctly, because it cannot be sorted. We again ended up with two different statements assigning values to the variable y. This problem can be fixed easily as follows:

```
when x < x1 or x > x2
   y = if x < 0 then y1 else y2;
end when;
```

Here, y assumes a new value if and only if either x becomes smaller than x_1 or if x becomes larger than x_2. The new value of y will be y_1, if x is at that time smaller than 0, else y assumes a value of y_2.

9.8 The Switch Equation

Let us now try to describe the electrical switch of Fig. 9.23.

When the switch is *open*, the current flowing through it is zero. When it is *closed*, the voltage across it is zero.

An elegant way to describe the switch properties in Dymola using a single statement would be:

```
0 = if switch == open then i else u;
```

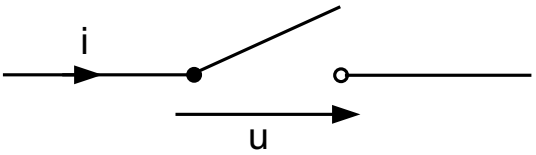

FIGURE 9.23. Electrical switch.

Let us convert the *if*–statement to an equivalent algebraic statement. To this end, we introduce an integer variable, m_o, with the following values:

switch	m_o
open	1
closed	0

Using the new variable m_o, we can rewrite the switch equation as follows:

$$0 = m_o \cdot i + (1 - m_o) \cdot u \tag{9.34}$$

The algebraic switch equation can be made causal in two different ways:

$$i = \frac{m_o - 1}{m_o} \cdot u \tag{9.35a}$$

$$u = \frac{m_o}{m_o - 1} \cdot i \tag{9.35b}$$

Unfortunately, neither of these two equations will work correctly in both switch positions. Equation (9.35a) will lead to a division by zero, whenever the switch closes, whereas Eq.(9.35b) will lead to a division by zero, whenever the switch opens.

The switch equation confronts us with a new problem. The correct computational causality of the switch equation depends on the numerical value of a parameter. In the given example, it depends on the numerical value of m_o.

In previous chapters, we have learnt that the computational causality of all equations is fixed, except for those that show up inside an *algebraic loop*.

Hence we may postulate that:

Switch equations must always be placed inside algebraic loops.

Let us illustrate this concept by means of a simple circuit example, as shown in Fig. 9.24.

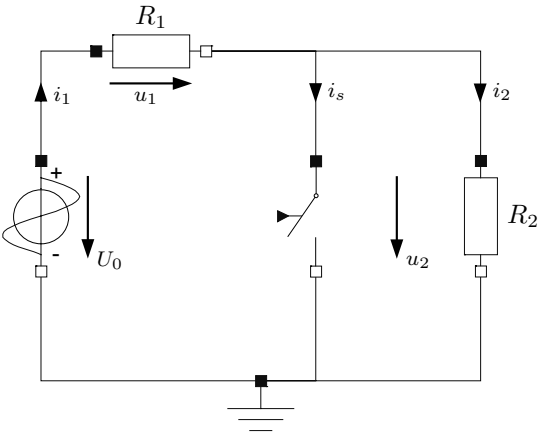

FIGURE 9.24. Electrical circuit containing a switch.

The circuit operates correctly in both switch positions. If the switch is open, the resistor across the voltage source assumes a value of $R_1 + R_2$, otherwise it assumes a value of R_1 only.

We can read out the equations from this circuit:

$$U_0 = f(t) \tag{9.36a}$$
$$u_1 = R_1 \cdot i_1 \tag{9.36b}$$
$$u_2 = R_2 \cdot i_2 \tag{9.36c}$$
$$U_0 = u_1 + u_2 \tag{9.36d}$$
$$i_1 = i_s + i_2 \tag{9.36e}$$
$$0 = m_o \cdot i_s + (1 - m_o) \cdot u_2 \tag{9.36f}$$

The structure digraph of this equation system is shown on Fig. 9.25.

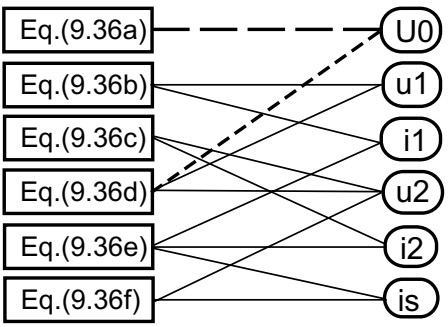

FIGURE 9.25. Partially causalized structure digraph of switching circuit.

The equation system indeed contains an algebraic loop in five equations

and five unknowns, and as expected, the switch equation shows up inside the algebraic loop.

This time around, we didn't use our normal heuristics for choosing a suitable tearing structure. We want our switch equation to serve as the residual equation, solving it for whichever variable works better. In the given example, we chose u_2 as the tearing variable, since this allowed us to causalize all remaining equations. The resulting set of causal equations is:

$$U_0 = f(t) \tag{9.37a}$$
$$i_2 = \frac{1}{R_2} \cdot u_2 \tag{9.37b}$$
$$u_1 = U_0 - u_2 \tag{9.37c}$$
$$i_1 = \frac{1}{R_1} \cdot u_1 \tag{9.37d}$$
$$i_s = i_1 - i_2 \tag{9.37e}$$
$$u_2 = \frac{m_o}{m_o - 1} \cdot i_s \tag{9.37f}$$

where Eq.(9.37f) is the residual equation, and u_2 serves as the tearing variable.

Using the variable substitution method, we find the following replacement equation for the residual equation:

$$u_2 = \frac{m_o \cdot R_2}{m_o \cdot (R_1 + R_2) + (m_o - 1) \cdot R_1 \cdot R_2} \cdot U_0 \tag{9.38}$$

Equation(9.38) is indeed the correct equation in both switch positions, since, when the switch is *closed*, i.e., $m_o = 0$:

$$u_2 = 0 \tag{9.39}$$

and when the switch is *open*, i.e., $m_o = 1$:

$$u_2 = \frac{R_2}{R_1 + R_2} \cdot U_0 \tag{9.40}$$

No division by zero is obtained in either of the two switch positions.

9.9 Ideal Diodes and Parameterized Curve Descriptions

Ideal diodes are ideal electrical switches complemented by an internal logic for determining the switch position. An ideal diode closes its switch, when

the voltage across the diode from the anode to the cathode becomes positive, and it opens its switch again, when the current through the diode passes through zero, if at that time the voltage across the diode is negative.

An ideal diode can be modeled in Dymola as follows:

$0 = m_o \cdot i_d + (1 - m_o) \cdot u_d$;
$m_o =$ **if** $u_d <= 0$ **and not** $i_d > 0$ **then** 1 **else** 0;

A yet more compact way to describe this model would be:

$0 =$ **if** *OpenSwitch* **then** i_d **else** u_d;
OpenSwitch $= u_d <= 0$ **and not** $i_d > 0$;

OpenSwitch is here a Boolean variable, the value of which is computed in the above Boolean expression. If *OpenSwitch* is *true*, the switch is considered *open*.

The latest example exhibits a third way for encoding state–event descriptions, beside from the previously introduced *if*–statements and *when*–statements. Any Boolean function of real–valued variables is automatically converted to a state–event description by Dymola's model compiler.

In reality, this is even the *only* way to produce state–event descriptions, as Dymola extracts the conditions of *if*– and *when*–statements into separate Boolean statements prior to expanding them.

How are Boolean functions of real–valued variables converted to zero–crossing functions? In the simplest cases, such as:

$$B_1 = x > x_2 \tag{9.41}$$

i.e., cases in which the Boolean expression is formed by a single relational operator, the conversion is trivial, as B_1 is almost in the correct form already. The corresponding zero–crossing function can be written as:

$$f_1 = x - x_2 \tag{9.42}$$

The case:

when $x < x_1$ **or** $x > x_2$
  $y =$ **if** $x < 0$ **then** y_1 **else** y_2;
end when;

is a bit more difficult to handle. The condition of the *when*–statement gets extracted into the Boolean function:

$$B_2 = x < x_1 \vee x > x_2 \tag{9.43}$$

which gets then converted to the following zero–crossing function:

$$f_2 = \textbf{if } B_2 \textbf{ then } 1 \textbf{ else } -1 \tag{9.44}$$

Whenever B_2 switches from *true* to *false* or vice–versa, f_2 crosses through zero.

Unfortunately, f_2 is anything but a smooth function. In fact, the gradient of f_2 is zero everywhere except at the zero crossing itself, where it is infinite. Thus, no higher–order method, such as *cubic interpolation* or *inverse Hermite interpolation* can be used on such a zero–crossing function. Only first–order methods, such as the *Regula Falsi* or the *Golden Section* method can be used, and of those, even only the Golden Section method can be used efficiently.

A better solution would have been to generate two separate zero–crossing functions:

$$f_{2a} = x - x_1 \tag{9.45a}$$

$$f_{2b} = x - x_2 \tag{9.45b}$$

that are both being associated with the same event action:

$$y = \textbf{if } x < 0 \textbf{ then } y_1 \textbf{ else } y_2 \tag{9.46}$$

The Dymola user can enforce that the model is being translated in this fashion by employing a slightly different model syntax:

when { $x < x_1$ **,** $x > x_2$ **}**
  $y =$ **if** $x < 0$ **then** y_1 **else** y_2;
end when;

Using this syntax, each of the set of conditions of the *when*–statement is converted independently to a separate zero–crossing function. All of these zero–crossing functions are associated with the same event action.

Unfortunately, even with the enhanced syntax, the problem:

$0 =$ **if** *OpenSwitch* **then** i_d **else** u_d;
OpenSwitch $= u_d <= 0$ **and not** $i_d > 0$;

cannot be converted to a set of smooth zero–crossing functions. The proposed technique works only in the case of a set of simple Boolean expressions that are connected by *or*–conditions. Another approach must thus be taken.

To this end, we shall apply a *parameterized curve description*, as advocated in [9.22]. Figure 9.26 displays the diode characteristic in the $i_d(u_d)$ plane.

The curve is *parameterized* by adding an additional variable, s, to the model, defined such that $s = u_d$ whenever the diode is blocking, and $s = i_d$, whenever the diode is conducting. This allows us to program a smooth zero–crossing function in terms of the newly introduced variable s:

$u_d =$ **if** *OpenSwitch* **then** s **else** 0;
$i_d =$ **if** *OpenSwitch* **then** 0 **else** s;

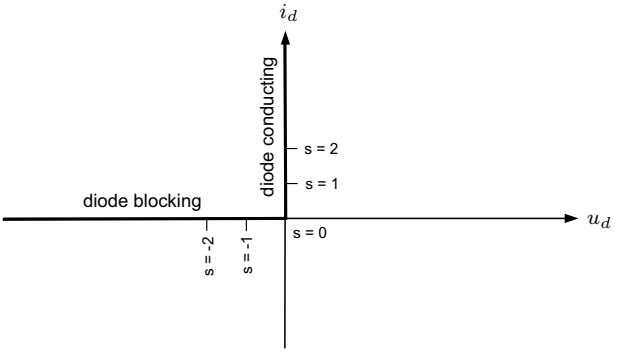

FIGURE 9.26. Diode characteristic.

$$OpenSwitch = s < 0;$$

This is how the ideal diode has been modeled in Dymola's standard electrical library.

An algebraic version of that model can be written as:

$$u_d = m_o \cdot s;$$
$$i_d = (1 - m_o) \cdot s;$$
$$m_o = \textbf{if } s < 0 \textbf{ then } 1 \textbf{ else } 0;$$

which is the version that we shall work with here, as these equations are easier to analyze.

Let us illustrate the use of the ideal diode model by means of the simple half–way rectifier circuit of Fig.9.27.

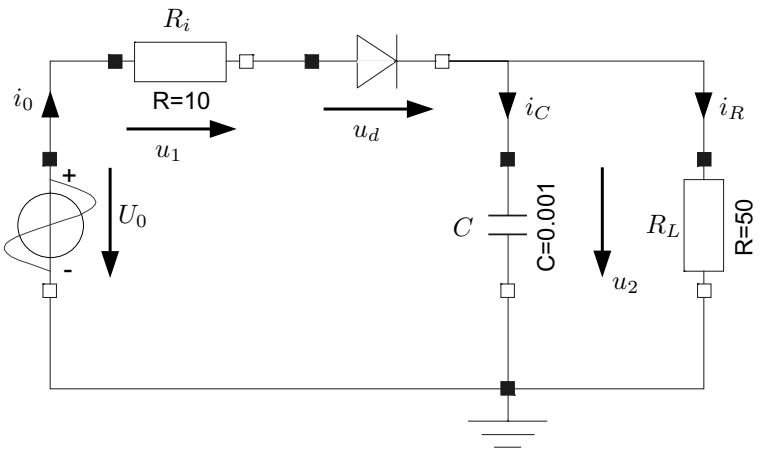

FIGURE 9.27. Half–way rectifier circuit.

We can read the equations from that circuit:

$$u_0 = f(t) \tag{9.47a}$$
$$u_1 = R_i \cdot i_0 \tag{9.47b}$$
$$u_2 = R_L \cdot i_R \tag{9.47c}$$
$$i_C = C \cdot \frac{du_2}{dt} \tag{9.47d}$$
$$u_0 = u_1 + u_d + u_2 \tag{9.47e}$$
$$i_0 = i_C + i_R \tag{9.47f}$$
$$u_d = m_o \cdot s \tag{9.47g}$$
$$i_0 = (1 - m_o) \cdot s \tag{9.47h}$$

The partially causalized structure digraph of this equation system is shown in Fig. 9.28. We ended up with an algebraic loop in four equations and four unknowns. The switch equations are contained within the algebraic loop.

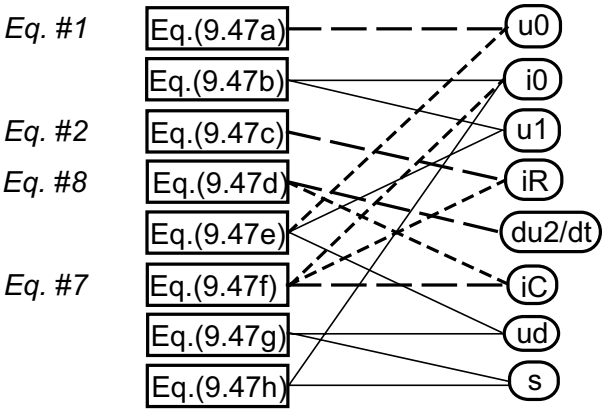

FIGURE 9.28. Partially causalized structure digraph.

We now must choose a suitable tearing structure. Once again, we won't use our normal heuristics. Instead, we want to make sure that the variable s is being selected as the tearing variable. We choose one of the two switch equations, e.g. Eq.(9.47h), as the corresponding residual equation.

The completely causalized structure digraph of this equation system is shown in Fig. 9.29.

Thus, the horizontally and vertically sorted equations can be written as:

$$u_0 = f(t) \tag{9.48a}$$

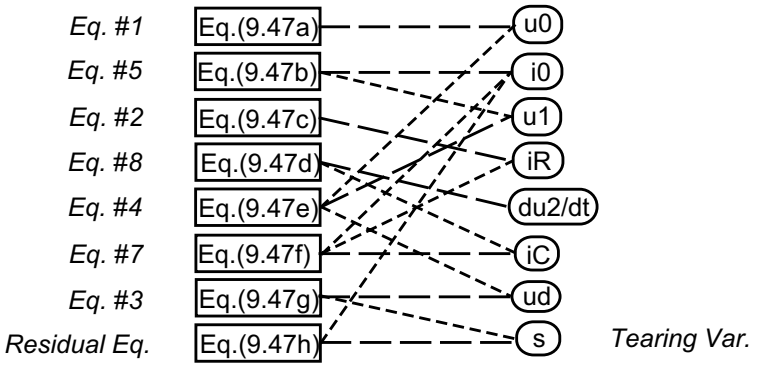

FIGURE 9.29. Completely causalized structure digraph.

$$i_R = \frac{1}{R_L} \cdot u_2 \tag{9.48b}$$

$$u_d = m_o \cdot s \tag{9.48c}$$

$$u_1 = u_0 - u_d - u_2 \tag{9.48d}$$

$$i_0 = \frac{1}{R_i} \cdot u_1 \tag{9.48e}$$

$$s = \frac{1}{1 - m_o} \cdot i_0 \tag{9.48f}$$

$$i_C = i_0 - i_R \tag{9.48g}$$

$$\frac{du_2}{dt} = \frac{1}{C} \cdot i_C \tag{9.48h}$$

where Eq.(9.48f) is the residual equation, with s having been chosen as the tearing variable.

Using the substitution technique, we find a replacement equation for the residual equation:

$$s = \frac{1}{m_o + (1 - m_o) \cdot R_i} \cdot (u_0 - u_2) \tag{9.49}$$

which is correct in both switch positions.

The following equation system results:

$$u_0 = f(t) \tag{9.50a}$$

$$i_R = \frac{1}{R_L} \cdot u_2 \tag{9.50b}$$

$$s = \frac{1}{m_o + (1 - m_o) \cdot R_i} \cdot (u_0 - u_2) \tag{9.50c}$$

$$u_d = m_o \cdot s \tag{9.50d}$$

$$u_1 = u_0 - u_d - u_2 \tag{9.50e}$$

$$i_0 = \frac{1}{R_i} \cdot u_1 \tag{9.50f}$$

$$i_C = i_0 - i_R \tag{9.50g}$$

$$\frac{du_2}{dt} = \frac{1}{C} \cdot i_C \tag{9.50h}$$

which can be simulated without any difficulties using any numerical integration algorithm with a root solver.

There is only a single zero–crossing function:

$$f = s \tag{9.51}$$

with the associated event action:

event *Toggle*
  $m_o := 1 - m_o$;
end *Toggle*;

The correct initial value of the discrete state variable, m_o, is assigned to that variable in an appropriate initialization section of the simulation program.

The voltage across the capacitor is shown in Fig. 9.30 as a function of time.

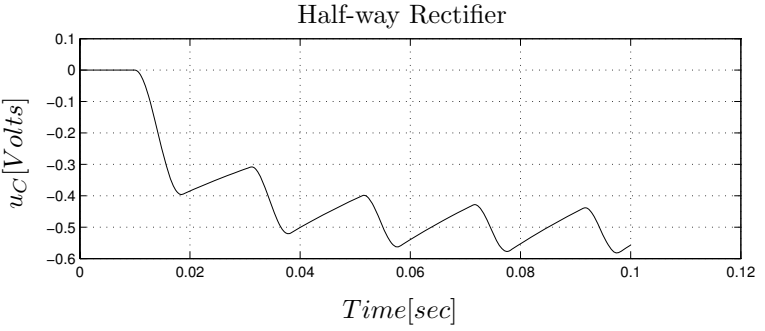

FIGURE 9.30. Voltage across capacitor of half–way rectifier circuit.

The default algorithm used in Dymola is DASSLRT, an implementation of the well–known DASSL algorithm supplemented with a root solver.

9.10 Variable Structure Models

Let us repeat the previous analysis for the slightly different circuit of Fig. 9.31.

The following set of equations characterizes this circuit:

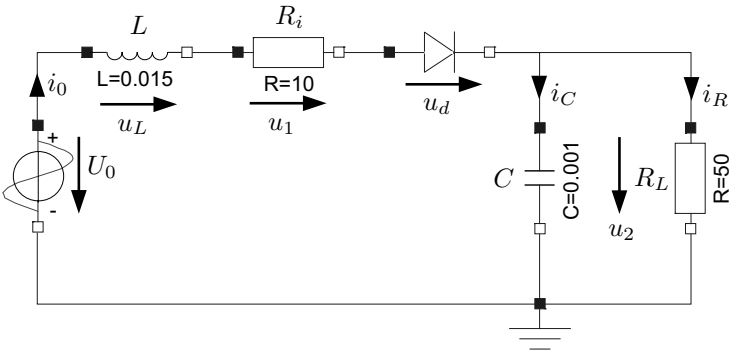

FIGURE 9.31. Half–way rectifier circuit with line inductance.

$$u_0 = f(t) \tag{9.52a}$$
$$u_1 = R_i \cdot i_0 \tag{9.52b}$$
$$u_2 = R_L \cdot i_R \tag{9.52c}$$
$$i_C = C \cdot \frac{du_2}{dt} \tag{9.52d}$$
$$u_L = L \cdot \frac{di_0}{dt} \tag{9.52e}$$
$$u_0 = u_L + u_1 + u_d + u_2 \tag{9.52f}$$
$$i_0 = i_C + i_R \tag{9.52g}$$
$$u_d = m_o \cdot s \tag{9.52h}$$
$$i_0 = (1 - m_o) \cdot s \tag{9.52i}$$

The structure digraph is shown in Fig. 9.32.

There is no algebraic loop. All equations have fixed causality. The causal equations are:

$$u_0 = f(t) \tag{9.53a}$$
$$u_1 = R_i \cdot i_0 \tag{9.53b}$$
$$i_R = \frac{1}{R_L} \cdot u_2 \tag{9.53c}$$
$$s = \frac{1}{1 - m_o} \cdot i_0 \tag{9.53d}$$
$$i_C = i_0 - i_R \tag{9.53e}$$
$$u_d = m_o \cdot s \tag{9.53f}$$
$$u_L = u_0 - u_1 - u_d - u_2 \tag{9.53g}$$
$$\frac{du_2}{dt} = \frac{1}{C} \cdot i_C \tag{9.53h}$$

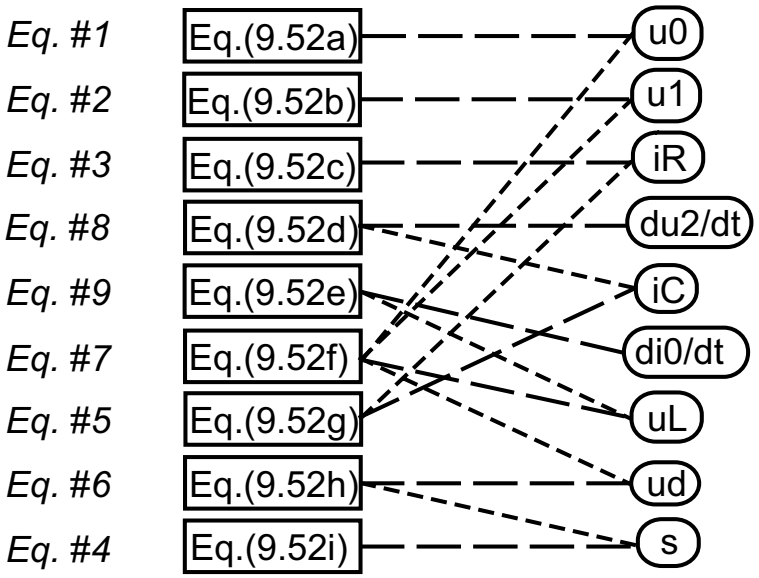

FIGURE 9.32. Causalized structure digraph of half–way rectifier circuit with line inductance model.

$$\frac{di_0}{dt} = \frac{1}{L} \cdot u_L \tag{9.53i}$$

These equations unfortunately cannot be simulated, since Eq.(9.53d) leads to a division by zero, as soon as the switch opens.

What happened? The current through the inductor is a state variable. Thus, the inductor computes the current i_0, which means that the causality of the diode is fixed. The diode has no choice but to compute the voltage u_d.

If we replace the diode by a manual switch, we see at once what happens. If we try to open the switch, while current is flowing through it, we'll draw an arc, because the current through the inductance cannot go instantly to zero. The arc can be modeled as a nonlinear resistor, the value of which increases, as the gap widens. This resistance drives the current to zero. Yet, this effect was not included in the model equations.

With a diode, this cannot happen, as the diode always opens at the moment, when the current passes through zero. Yet, our model doesn't know this. Since the logic for when the diode switch opens or closes is not contained in the continuous model equations, but forms part of the event description, the continuous model equations are identical in the case of the diode and the manual switch.

Dymola tackles this problem by offering in its standard electrical library a *leaky diode* model, as shown in Fig. 9.33.

The leaky diode can be modeled using the equations:

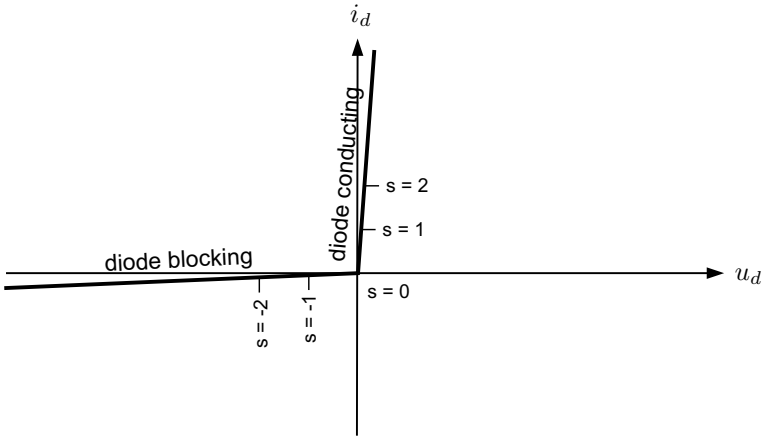

FIGURE 9.33. Leaky diode characteristic.

$u_d = \textbf{if } \mathit{OpenSwitch} \textbf{ then } s \textbf{ else } R_0 \cdot s;$
$i_d = \textbf{if } \mathit{OpenSwitch} \textbf{ then } G_0 \cdot s \textbf{ else } s;$
$\mathit{OpenSwitch} = s < 0;$

or formulated algebraically:

$u_d = [m_o + (1 - m_o) \cdot R_0] \cdot s;$
$i_d = [m_o \cdot G_0 + (1 - m_o)] \cdot s;$
$m_o = \textbf{if } s < 0 \textbf{ then } 1 \textbf{ else } 0;$

R_0 is the resistance of the wires connected to the switch, when the switch is closed, and G_0 is the conductance of the air in the gap, while the switch is open.

The leaky diode model doesn't change the causalities of the equation system, i.e., the structure digraph of the model using the leaky diode is exactly the same as that using the ideal diode. However, the leaky diode avoids the division by zero.

The causal equations of the model using the leaky diode are:

$$u_0 = f(t) \tag{9.54a}$$
$$u_1 = R_i \cdot i_0 \tag{9.54b}$$
$$i_R = \frac{1}{R_L} \cdot u_2 \tag{9.54c}$$
$$s = \frac{1}{m_o \cdot G_0 + (1 - m_o)} \cdot i_0 \tag{9.54d}$$
$$i_C = i_0 - i_R \tag{9.54e}$$
$$u_d = [m_o + (1 - m_o) \cdot R_0] \cdot s \tag{9.54f}$$

$$u_L = u_0 - u_1 - u_d - u_2 \tag{9.54g}$$

$$\frac{du_2}{dt} = \frac{1}{C} \cdot i_C \tag{9.54h}$$

$$\frac{di_0}{dt} = \frac{1}{L} \cdot u_L \tag{9.54i}$$

This model is valid in both switch positions, i.e., it can be simulated. Unfortunately, whenever the original model containing an ideal diode exhibits a division by zero, the new model containing a leaky diode becomes very stiff. The degree of stiffness is directly related to the values of the two leakage parameters, R_0 and G_0. The smaller the leakage parameters are chosen, the stiffer the model will become. Hence we would prefer to use the ideal model, if we could.

What is so special about this model? When the switch is closed, i.e., while the diode is conducting, the model exhibits second–order dynamics. However, once the switch opens, i.e., while the diode blocks the current, we are faced with only first–order dynamics. The inductor does not contribute to the dynamics in that case.

We call a model that exhibits different structural properties, such as a varying number of differential equations depending on the position of some switches a *variable structure model*.

Variable structure systems are very common, e.g. in mechanical engineering. All systems involving clutches are by their very nature variable structure systems. In electrical engineering, most switching power converters are variable structure systems.

The way, the equations of our system were formulated, Eqs.(9.52a–i), it doesn't look like these equations contain a *structural singularity* though. There is no constraint equation to be found. The singularity looks to be *parametric* in nature, thus the Pantelides algorithm [9.23] cannot be applied to solve it.

9.11 Mixed–mode Integration

One way to tackle this problem, while preserving the use of an ideal diode model, is to relax the causality on the inductor, by inlining the integrator that is associated with it. This approach was first proposed in [9.18].

An approach to simulation by applying different integration algorithms to different integrators contained in the model is called simulation by mixed–mode integration [9.24].

The system equations now take the form:

$$u_0 = f(t) \tag{9.55a}$$

$$u_1 = R_i \cdot i_0 \tag{9.55b}$$

$$u_2 = R_L \cdot i_R \tag{9.55c}$$

$$i_C = C \cdot \frac{du_2}{dt} \tag{9.55d}$$

$$i_0 = \mathrm{pre}(i_o) + \frac{h}{L} \cdot u_L \tag{9.55e}$$

$$u_0 = u_L + u_1 + u_d + u_2 \tag{9.55f}$$

$$i_0 = i_C + i_R \tag{9.55g}$$

$$u_d = m_o \cdot s \tag{9.55h}$$

$$i_0 = (1 - m_o) \cdot s \tag{9.55i}$$

The partially causalized structure digraph of this model is given in Fig. 9.34.

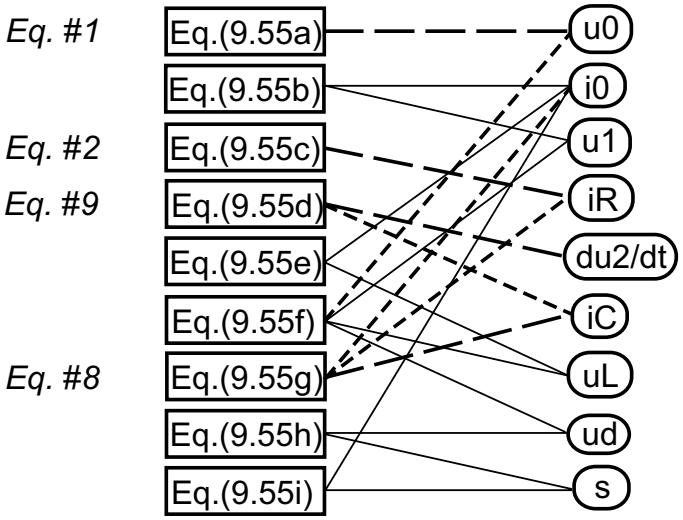

FIGURE 9.34. Partially causalized structure digraph of half–way rectifier circuit with inlined inductor.

Only four of the nine equations could be causalized directly. There now appeared an algebraic loop, which includes the switch equations.

We need to choose s as a tearing variable, because otherwise, the equation computing s will invariably contain either the term m_o or the term $(1-m_o)$ alone in the denominator, which consequently leads to a division by zero in one of the two switch positions.

We can choose either Eq.(9.55h) or Eq.(9.55i) as the associated residual equation. If we choose Eq.(9.55h) as the residual equation, we can causalize all of the remaining equations. Unfortunately, Eq.(9.55e) will in that case be solved for u_L, which we don't like, since it leaves h alone in the denominator.

Thus, we chose Eq.(9.55i) as the associated residual equation. The further causalized structure digraph is shown in Fig. 9.35.

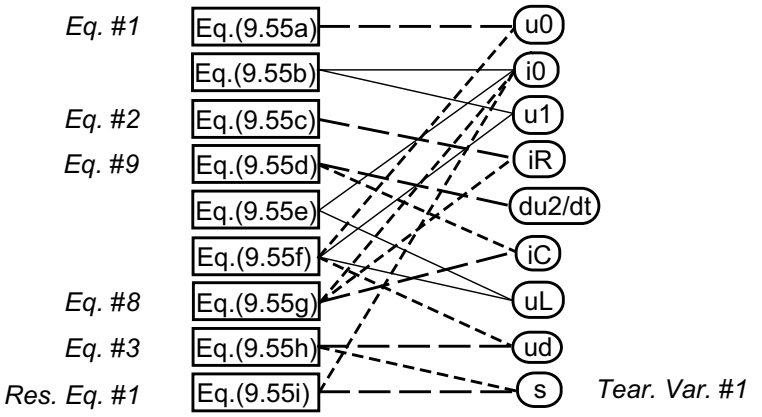

FIGURE 9.35. Partially causalized structure digraph of half–way rectifier circuit with inlined inductor.

There remains a second algebraic loop in three equations and three unknowns. This time, we choose Eq.(9.55e) as the new residual equation, and i_0 as the tearing variable. In this way, we can force the causality on the inlined integrator equation as well. The completely causalized structure digraph is shown in Fig. 9.36.

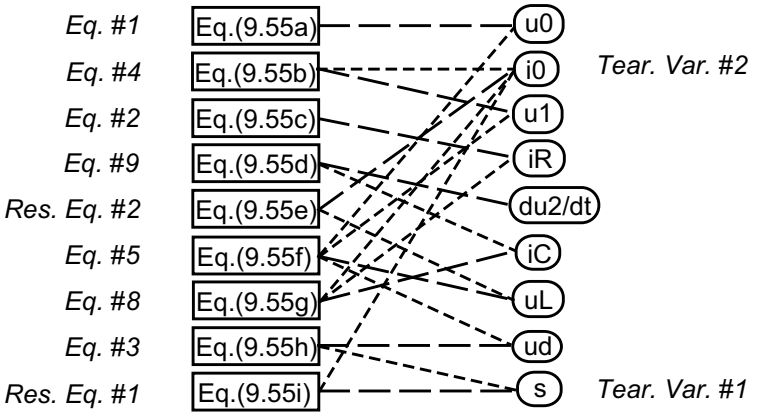

FIGURE 9.36. Completely causalized structure digraph of half–way rectifier circuit with inlined inductor.

The causalized equations can be read out of the structure digraph:

$$u_0 = f(t) \tag{9.56a}$$

$$i_R = \frac{1}{R_L} \cdot u_2 \tag{9.56b}$$

$$u_d = m_o \cdot s \tag{9.56c}$$
$$u_1 = R_i \cdot i_0 \tag{9.56d}$$
$$u_L = u_0 - u_1 - u_d - u_2 \tag{9.56e}$$
$$i_0 = \mathrm{pre}(i_o) + \frac{h}{L} \cdot u_L \tag{9.56f}$$
$$s = \frac{1}{1 - m_o} \cdot i_0 \tag{9.56g}$$
$$i_C = i_0 - i_R \tag{9.56h}$$
$$\frac{du_2}{dt} = \frac{1}{C} \cdot i_C \tag{9.56i}$$

Using the variable substitution technique, we find replacement equations for the two residual equations. The final set of horizontally and vertically sorted equations presents itself as follows:

$$u_0 = f(t) \tag{9.57a}$$
$$i_R = \frac{1}{R_L} \cdot u_2 \tag{9.57b}$$
$$s = \frac{L \cdot \mathrm{pre}(i_o) + h \cdot (u_0 - u_2)}{h \cdot m_o + (L + h \cdot R_i) \cdot (1 - m_o)} \tag{9.57c}$$
$$u_d = m_o \cdot s \tag{9.57d}$$
$$i_0 = (1 - m_o) \cdot s \tag{9.57e}$$
$$u_1 = R_i \cdot i_0 \tag{9.57f}$$
$$u_L = u_0 - u_1 - u_d - u_2 \tag{9.57g}$$
$$i_C = i_0 - i_R \tag{9.57h}$$
$$\frac{du_2}{dt} = \frac{1}{C} \cdot i_C \tag{9.57i}$$

Let us analyze this set of equations a bit further. The only potentially dangerous equation is Eq.(9.57c). Let us discuss, how this equation behaves in the two switch positions.

If the switch is closed, $m_o = 0$, Eq.(9.57c) degenerates to:

$$s = \frac{L \cdot \mathrm{pre}(i_o) + h \cdot (u_0 - u_2)}{L + h \cdot R_i} \tag{9.58}$$

which is completely harmless for all values of the step size, h.

If the switch is open, $m_o = 1$, Eq.(9.57c) degenerates to:

$$s = \frac{L}{h} \cdot \mathrm{pre}(i_o) + u_0 - u_2 \tag{9.59}$$

This equation is correct for all values of the step size, h, if switching occurs at a moment, when the current, i_0, goes through zero, as this will always be

the case for a diode. However, if switching occurs for any other value of i_0, only one step will be incorrect, since during that first step, the current i_0 will be reduced to zero due to Eq.(9.57e). Thus, already one step later, the solution is again accurate. There is no stiffness problem using this approach.

9.12 State Transition Diagrams

Let us now return to the discussion of friction phenomena, an important application of discontinuous models in mechanical engineering.

Before a possible general model for the friction element can be presented, the friction phenomenon needs to be carefully analyzed. According to Fig.9.20, the friction force is a known applied force if the velocity v is different from zero. In that situation, the computational causality of the friction model is such that the velocity is an input to the model, whereas the friction force is its output.

When the velocity becomes zero, the two bodies, between which the friction force is acting, become stuck. In this situation, the model changes its structure: A new equation, $v = 0.0$, and a new unknown force, F_c, are added to the model. The constraint force F_c is determined such that the new constraint on the velocity, $v = 0.0$, is always met.

This is a new situation as compared to the electrical switch, because the electrical switch toggles between two different equations one of which is always active. Thus, the number of equations remains the same. In contrast, the friction element adds one equation and one variable to the model, when v becomes 0, and removes them again, when abs(F_c) becomes larger than the threshold value F_s.

Simulation environments do not usually allow to add/remove variables during integration. Therefore, a dummy equation is added, which becomes active, when the constraint equation, $v = 0.0$, is removed. The dummy equation is used to provide a unique –but arbitrary– value for F_c during sliding motion. For example, $F_c = 0.0$ is as good a value as any.

The friction force F can thus be defined through the following equations:

$$\begin{aligned} F &= \textbf{if } v > 0 \textbf{ then } c_f \cdot v + F_d \textbf{ else} \\ &\quad\ \textbf{if } v < 0 \textbf{ then } c_f \cdot v - F_d \textbf{ else } F_c \\ 0 &= \textbf{if } \mathit{Sticking} \textbf{ then } v \textbf{ else } F_c \end{aligned}$$

The third equation is our meanwhile well–known *switch equation*.

The model is so simple, it looks like magic . . . and it also works like magic, i.e., it doesn't. In a Newtonian world, it is not sufficient to describe how the rabbit is being pulled out of the magician's hat. We also need to describe how it got into the hat in the first place, at which time, unfortunately, the magic is gone.

There are two problems with the above model. First, we haven't come up yet with an equation for the discrete state variable *Sticking*. Evidently,

it won't do to say that:

$$Sticking = v == 0 \tag{9.60}$$

as this would simply state that whenever v equals zero, then v equals zero, which is undoubtedly a true statement, but unfortunately, it isn't a very useful one.

The second problem with this model is that the *then*–branch of the switch equation is constrained, since the velocity v is a state variable. Thus, the causality of the switch equation is fixed, which invariably leads to a division by zero in one of the two switch positions.

Let us tackle the latter problem first. We already know one possible solution to this problem. We could relax the causality on the velocity, v, by inlining the integrator that integrates the acceleration, a, into the velocity, v. Yet, there is a better way.

While in the sticking position, the velocity, v, remains constantly zero. Thus, also the acceleration, a, must remain constantly zero. We can thus replace the former switch equation in the velocity, v, by a modified switch equation in the acceleration, a, as follows:

$$\begin{aligned} F &= \textbf{if } v > 0 \textbf{ then } c_f \cdot v + F_d \textbf{ else} \\ &\quad\ \textbf{if } v < 0 \textbf{ then } c_f \cdot v - F_d \textbf{ else } F_c \\ 0 &= \textbf{if } Sticking \textbf{ then } a \textbf{ else } F_c \end{aligned}$$

This looks like a generalization of the Pantelides algorithm [9.23]. We seem to have partially differentiated the switch equation. Unfortunately, this technique rarely works. The Pantelides algorithm can only be generalized to the case of *conditional index changes* modeled by means of switch equations, if either both branches of the *if*–statement formulating the switch equation are constrained, or if the unconstrained branch is unimportant.

In the case of the friction model, the *then*–branch of the switch equation is constrained, as it is a function of state variables only, whereas the *else*–branch is unimportant. While the model is not sticking, we don't care what value the variable F_c assumes. Thus, there is no need to differentiate the *else*–branch of the switch equation simultaneously with the *then*–branch.

There is still a small problem with this formulation though. Since the friction model enters its *Sticking* region when the velocity passes through zero, the velocity may numerically not be exactly equal to zero, after the model entered its "*Sticking*" region. Therefore, the position, x, may slowly drift away.

This problem can be easily fixed by adding:

$$\begin{aligned} F &= \textbf{if } v > 0 \textbf{ then } c_f \cdot v + F_d \textbf{ else} \\ &\quad\ \textbf{if } v < 0 \textbf{ then } c_f \cdot v - F_d \textbf{ else } F_c \\ 0 &= \textbf{if } Sticking \textbf{ then } a \textbf{ else } F_c \\ &\textbf{when } Sticking \textbf{ then} \\ &\quad reinit(v, 0); \\ &\textbf{end when}; \end{aligned}$$

to the model. Thus, when the friction model enters its *Sticking* region, the velocity, v, is explicitly re–initialized to 0.

Let us now tackle the other problem. We haven't defined yet, how the discrete state variable, *Sticking*, is computed by the model. To this end, we need to define, how the switching between the sliding and the sticking phases takes place.

It is advantageous to split the friction force law into the following five different regions:

region:		**region conditions:**	
Forward	:	$v > 0$	**and** $F = c_f \cdot v + F_d$
StartForward	:	$v = 0$ **and** $a > 0$	**and** $F = +F_d$
Sticking	:	$v = 0$ **and** $a = 0$	**and** $F \in [-F_s, +F_s]$
StartBackward	:	$v = 0$ **and** $a < 0$	**and** $F = -F_d$
Backward	:	$v < 0$	**and** $F = c_f \cdot v - F_d$

Regions *Forward* and *Backward* describe the sliding phase and are defined by a non–zero velocity. Region *Sticking* denotes the sticking phase and is defined by identically vanishing velocity and acceleration. Regions *StartForward* and *StartBackward* define the transition from sticking to sliding. These regions are characterized by a zero velocity. The difference to the sticking phase is that the acceleration is no longer fixed to zero. The above five regions cannot be encoded directly, because the equality relation "=" appears in the definition. It is not meaningful to test computed real–valued variables for being equal to zero.

Hence an indirect approach will be used. The switching between the five regions is described by a *deterministic finite state machine (DFSM)* [9.1]. The *state transition diagram* of the DFSM was shown earlier in this chapter. It is repeated here.

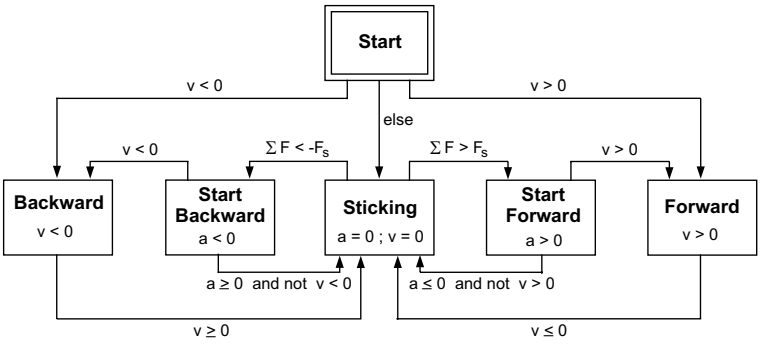

FIGURE 9.37. State transition diagram of friction characteristic.

The DFSM has six states, corresponding to the five regions of the model and a *Start* state. Starting from any one state of the DFSM and using one of the mutually exclusive conditions, a new state of the DFSM is reached

in an unambiguous fashion. None of the switching conditions contains the equality relation.

A valid Dymola code can be easily derived from a DFSM by defining a boolean variable (a discrete state variable) for every state of the DFSM and by encoding the state transitions leading into or out of each state as boolean expressions determining the next value of that state.

```
F = if Forward       then c_f · v + F_d else
    if Backward      then c_f · v − F_d else
    if StartForward  then +F_d         else
    if StartBackward then −F_d         else F_c;

0 = if Sticking or Start then a else F_c;

Forward       = pre(Start)       and v > 0 or
                pre(StartForward) and v > 0 or
                pre(Forward)      and not v <= 0;

Backward      = pre(Start)       and v < 0 or
                pre(StartBackward) and v < 0 or
                pre(Backward)     and not v >= 0;

StartForward  = pre(Sticking)    and F_c > +F_s or
                pre(StartForward) and not
                (v > 0 or a <= 0 and not v > 0);

StartBackward = pre(Sticking)    and F_c < −F_s or
                pre(StartBackward) and not
                (v < 0 or a >= 0 and not v < 0);

Sticking      = not (Start or
                Forward or StartForward or
                Backward or StartBackward);

when Sticking and not Start then
    reinit(v, 0);
end when;
```

Comparing this Dymola model with the DFSM of Fig.9.37, it can be seen that the translation of one into the other is systematic and quite straightforward.

This model can be simulated. Unfortunately, it is characterized by fairly complicated switching conditions that lead to zero–crossing functions that aren't smooth. Let us see, whether this situation can be rectified.

To this end, we shall employ the parameterized curve description technique once again. Figure 9.38 shows a slightly simplified friction characteristic that has been parameterized in similar ways as with the diode characteristic introduced earlier in this chapter.

The curve parameter is defined as follows:

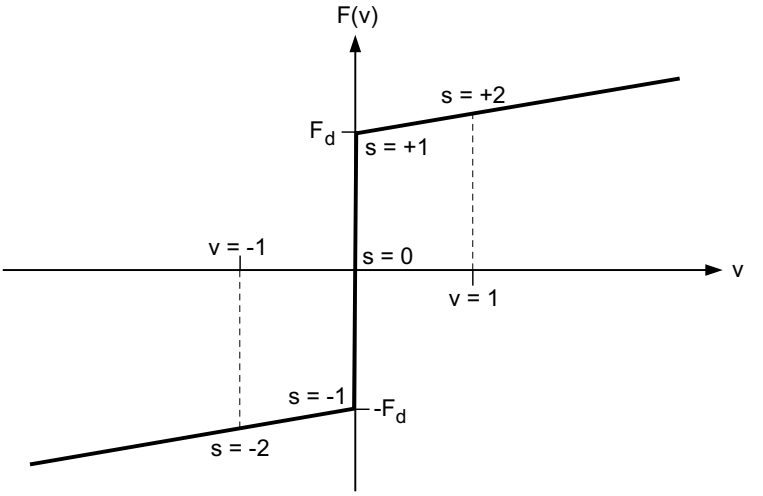

FIGURE 9.38. Simplified friction characteristic with curve parameterization.

region	s
forward	$v + 1$
sticking	F/F_d
backward	$v - 1$

Curve parameters can be defined in any way that is most suitable. They don't have to be equidistantly spaced, and they can even adopt different units in different regions, as the example demonstrates. Using the new variable, s, we can define the simplified friction model as follows [9.22]:

$$
\begin{array}{l}
\mathit{Forward} \;\; = s > +1; \\
\mathit{Backward} = s < -1; \\
v = \textbf{if } \mathit{Forward} \;\; \textbf{then } s - 1 \qquad\qquad\quad \textbf{else} \\
\qquad \textbf{if } \mathit{Backward} \; \textbf{then } s + 1 \qquad\quad \textbf{else } 0; \\
F = \textbf{if } \mathit{Forward} \;\; \textbf{then } c_f \cdot (s-1) + F_d \; \textbf{else} \\
\qquad \textbf{if } \mathit{Backward} \textbf{then } c_f \cdot (s+1) - F_d \; \textbf{else } F_d \cdot s;
\end{array}
$$

This model is correct in the sense that it describes unambiguously our intentions of what the model is supposed to accomplish. Thus, we might expect that a decent model compiler would be capable of translating the model down to an event description that can be properly simulated.

Unfortunately, the Dymola model compiler, as it is currently implemented, is unable to do so. There are two problems with this model. Let us explain.

While the model operates in its *Forward* region, the velocity, v, is a state variable, thus can be assumed known. Hence the curve parameter, s, can be computed from the equation $s = v + 1$, and the friction force can be

obtained using the equation $F = c_f \cdot (s - 1) + F_d$.

What happens, when s becomes smaller than +1? The model is now entering its *Sticking* region. In this region, we have the equation; $v = 0$. Thus, the velocity, v, can no longer be treated as a known state variable, and we are confronted with a *conditional index change*. Somehow, we shall have to deal with this problem.

Let us now assume that the model is currently operating in its *Sticking* region. What happens, when s becomes larger than +1? The model is now entering its *Forward* region. In this region, we compute s using the equation $s = v + 1$, and since v was initialized to zero after the event, s returns immediately back to one. As a consequence, a new state event is triggered that throws the model right back into its *Sticking* region. Thus, the model is stuck in its *Sticking* region forever! This problem seems to be related to the narrow band problem encountered earlier, although it manifests itself a bit differently.

We can tackle the former problem using the same argumentation that had been used already in the previous model: If the velocity, v, is constantly equal to zero over a period of time, then also the acceleration, a, must be constantly equal to zero during that time period.

Thus, we can describe the *Sticking* region and its immediate surroundings by looking at the acceleration, rather than the velocity. This concept is illustrated in Fig. 9.39.

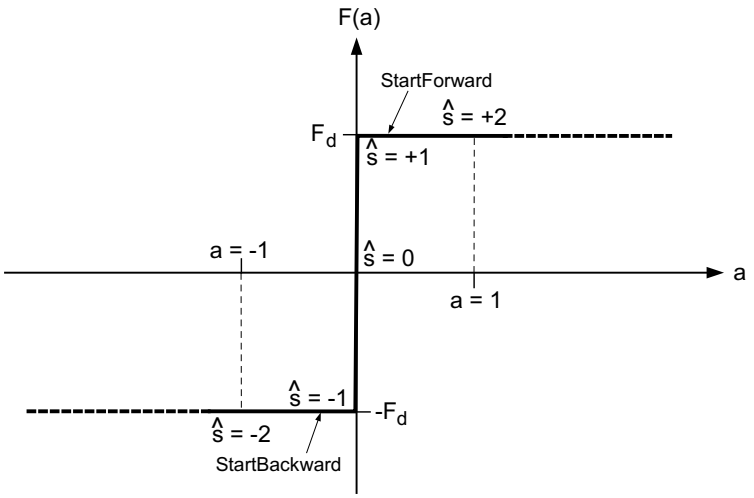

FIGURE 9.39. Sticking region of simplified friction characteristic with curve parameterization.

Since this is a different friction curve from the one shown before, the model uses a different parameter for its curve parameterization, $\hat{s}$. The model can be described using the same techniques introduced earlier:

$$
\begin{aligned}
&\mathit{StartForward} &&= \hat{s} > +1;\\
&\mathit{StartBackward} &&= \hat{s} < -1;\\
&a = \textbf{if } \mathit{StartForward} &&\textbf{ then } \hat{s} - 1 \textbf{ else}\\
&\quad \textbf{if } \mathit{StartBackward} &&\textbf{ then } \hat{s} + 1 \textbf{ else } 0;\\
&F = \textbf{if } \mathit{StartForward} &&\textbf{ then } +F_d \textbf{ else}\\
&\quad \textbf{if } \mathit{StartBackward} &&\textbf{ then } -F_d \textbf{ else } F_d \cdot \hat{s};
\end{aligned}
$$

We shall use a DFSM to describe the switching between the three main regions of the model, as illustrated in Fig. 9.40.

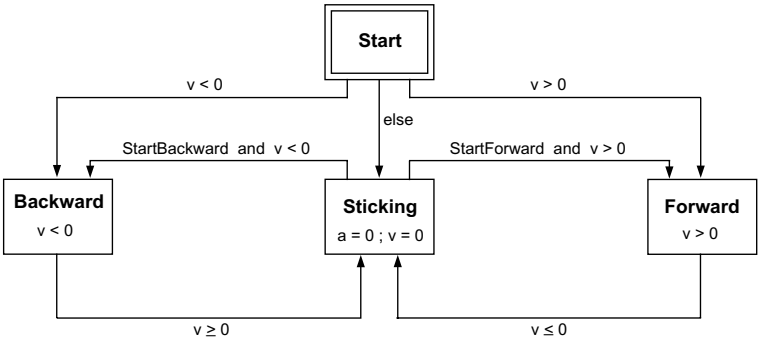

FIGURE 9.40. Deterministic finite state machine modeling the switching events of the simplified friction characteristic.

This is a much simplified version of the DFSM of Fig. 9.37 used by the earlier model. The new DFSM has only four instead of six discrete states (regions). The *StartForward* and *StartBackward* modes of operation are no longer considered separate regions. Instead, they are contained within the *Sticking* region model. They only represent different aspects of the *Sticking* region.

We shall not offer an encoding of the DFSM at this point, but instead, we shall leave this problem for one of the exercises at the end of this chapter.

Unfortunately, the simplified DFSM still contains two mixed switching conditions, describing the conditions under which the model leaves the *Sticking* region. These switching conditions prevent Dymola from generating smooth zero–crossing functions in those cases. Yet, the problem is not too damaging numerically, because these switchings occur always as an almost immediate result of a previous switching to one of the two transitory modes, *StartForward* and *StartBackward*, for which smooth zero–crossing functions in the curve parameter, $\hat{s}$, had been defined.

9.13 Petri Nets

We shall now demonstrate that it is always possible to decompose complex (combined) event conditions into sets of simple event conditions that consist of a single relational operator only. Thus, all zero–crossing functions can be made smooth. To this end, we shall introduce a new model description tool: the *Petri net*.

Petri nets [9.20] consist of two modeling elements: *places* and *transitions*. Places are holders of *tokens*. Each place maintains a discrete state variable that counts the number of tokens currently held by the place. Transitions connect places. When a transition *fires*, it takes some tokens out of places connected at its inputs, and places some new tokens at places connected at its outputs in accordance with some logic to be defined. A transition may fire, when an external firing condition is true, if the conditions concerning the necessary numbers of tokens held by its input places are true as well.

If one place feeds several transitions, additional logic may be required to determine firing preferences in the case of simultaneous events, i.e., in the case where the external firing conditions of several transitions become true simultaneously, because there may be enough tokens in the input place to fire one or the other of these transitions, but not all of them.

Many different dialects of Petri nets have been described in the literature [9.21].

Bounded Petri nets are Petri nets with capacity limitations imposed on its places. *Normal Petri nets* are Petri nets with a capacity limit of one imposed on each place. In a normal Petri net, the discrete state counting the number of tokens contained in a place can thus be represented as a Boolean state. If the state has a value of *true*, there is a token located at the place. If the state has a value of *false*, there is no token at the place.

Priority Petri nets resolve the ambiguity associated with multiple transitions being able to fire simultaneously by associating a prioritization scheme to these transitions.

Normal priority Petri nets (NPPNs) are normal Petri nets employing prioritization schemes in all of their transitions.

A NPPN place with two inputs and two outputs has been depicted in Fig. 9.41.

The place passes state information, s_i, to all neighboring transitions, and in turn receives firing information, f_i, back from these transitions.

The NPPN place could be governed by the following equations:

$$s_1 = \mathrm{pre}(p_1) \tag{9.61a}$$

$$s_2 = \mathrm{pre}(p_1) \textbf{ or } f_1 \tag{9.61b}$$

$$s_3 = \mathrm{pre}(p_1) \tag{9.61c}$$

$$s_4 = \mathrm{pre}(p_1) \textbf{ and not } f_3 \tag{9.61d}$$

$$p_1 = [\mathrm{pre}(p_1) \textbf{ and not } (f_3 \textbf{ or } f_4)] \textbf{ or } f_1 \textbf{ or } f_2 \tag{9.61e}$$

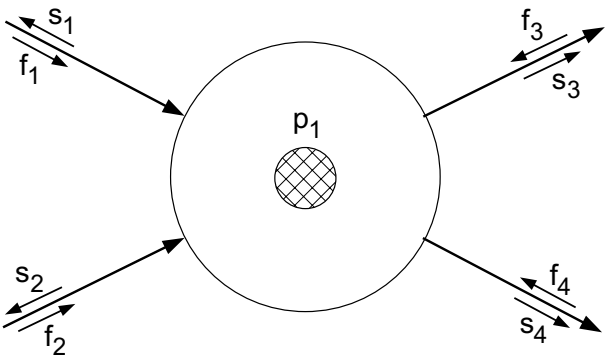

FIGURE 9.41. NPPN place with two inputs and two outputs.

The logic of these equations goes as follows. The place first provides the first input transition, t_1, with its state information. Transition t_1 needs to know this information, because, due to the single–token capacity limitation, it cannot fire, unless the place, p_1, is currently unoccupied. The place receives the firing information, f_1, back from transition t_1. If t_1 fires, it means that it is going to place a new token at p_1.

The place then provides the appropriate state information, s_2, to the second input transition, t_2. Transition t_2 is assigned a lower priority than transition t_1. Transition t_2 is not allowed to fire if either there is already a token at place p_1, or if the other input transition, t_1, decided to fire, because if both transitions were to fire simultaneously, they both would try to place a token at p_1, which would violate the imposed capacity limit of one.

The place then provides its state information to the first output transition, t_3. Transition t_3 is allowed to fire if a token is currently at p_1. If it fires, it will take the token away from place p_1.

The place then provides the appropriate state information, s_4, to the second output transition, t_4. Transition t_4 is assigned a lower priority than transition t_3. Transition t_4 is not allowed to fire, unless there is currently a token at place p_1 and transition t_3 has not decided to fire, because if both transitions were to fire simultaneously, they both would fight over who gets to remove the token from p_1.

Finally, the place must update its own state information. If there was a token at p_1 before, and neither of the two output transitions, t_3 or t_4, has taken it away, or, if one of the two input transitions, t_1 or t_2, has placed a new token at p_1, there will be a token at that place during the next cycle.

Let us now look at a transition with two input places and two output places. It has been depicted in Fig. 9.42.

The logic governing the transitions could be the following. The transition is allowed to fire along all of its connections, when the external firing

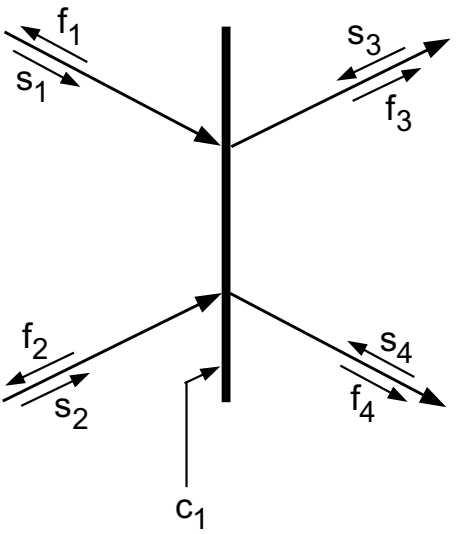

FIGURE 9.42. NPPN transition with two inputs and two outputs.

condition, c_1, is true, and if each of the input places holds a token (or more precisely, if the state information arriving from all of the input places is *true*), and if none of the output places holds a token (or more precisely, if the state information of none of the output places is *true*).

This logic can be described by the following set of equations:

$$\textit{fire} = c_1 \textbf{ and } s_1 \textbf{ and } s_2 \textbf{ and not } (s_3 \textbf{ or } s_4) \tag{9.62a}$$

$$f_1 = \textit{fire} \tag{9.62b}$$

$$f_2 = \textit{fire} \tag{9.62c}$$

$$f_3 = \textit{fire} \tag{9.62d}$$

$$f_4 = \textit{fire} \tag{9.62e}$$

DFSMs can be modeled as normal priority Petri nets with the additional constraints that there is only one token in the system that is initially located at the *Start* place. Furthermore, DFSMs map to NPPNs, in which each transition is associated with exactly one input place and one output place.

Let us model the DFSM of Fig. 9.37 as a Petri net. The corresponding NPPN representation is depicted in Fig. 9.43.

We immediately recognize what the external firing conditions, c_i, represent. These are the conditions that are associated with state transitions in the DFSM. Hence those are the edge–triggered Boolean variables associated with the zero–crossing functions.

What have we gained by this representation? In the past, we had many different discrete event blocks representing the actions to be taken, when one or the other of the zero–crossing functions triggered an event. This is

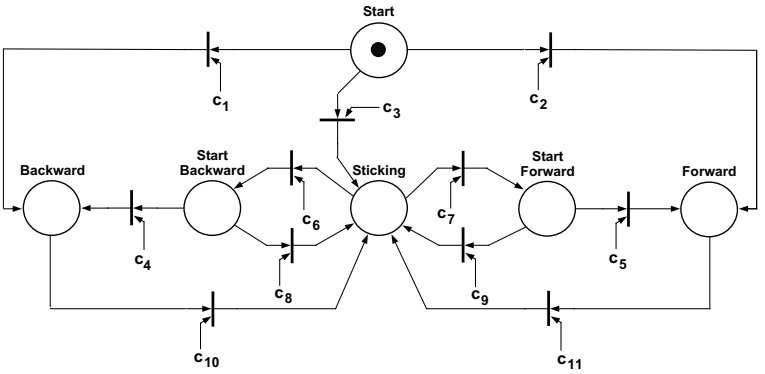

FIGURE 9.43. Petri net representation of friction characteristic.

no longer the case. All of the discrete equations governing both *places* and *transitions* are valid at every event, since they were formulated as functions of the current location of the tokens, i.e., they were functions of the discrete state that the system is currently operating in.

Thus, every discontinuous model, as complex as it may be, can be described by exactly three sets of equations. There are the implicitly defined algebraic and differential equations describing the continuous subsystem. There is the set of zero–crossing functions that are all evaluated in parallel, while the continuous subsystem is being simulated. If a state event is being triggered by one of them, an iteration (or interpolation) takes place to locate the event time as accurately as necessary. At that moment, the third set of simultaneous equations is being executed. These are the (possibly implicitly defined) algebraic and difference equations describing the discrete subsystem.

The discrete equations are executed iteratively, until no discrete state changes occur any longer. When this happens, we have found our new initial state, from which we can start the continuous simulation afresh.

A simulation model that has been compiled into this form, can be simulated in an organized and systematic fashion based on a synchronous data flow [9.22].

It may not be convenient for the end user of the modeling and simulation environment to describe his or her model in this fashion. Different application domains make use of different modeling formalisms that users are familiar with. It is the job of the *model compiler* to dissect the model description that the user supplies, and translate it down to sets of simultaneous equations that can be simulated without numerical difficulties.

As this book concerns itself with the set of algorithms underlying a powerful modeling and simulation environment, such as Dymola [9.12], we had to show step by step, how model equations need to be preconditioned, until they are finally in a form such that they can be simulated without

difficulties. Yet, it was no longer convenient to translate every model that we came across manually down to such a form.

Will the iteration on the simultaneous discrete equations always converge? If the model of a physical system is formulated correctly, the iteration should always converge, as our Newtonian world is deterministic in nature. Yet, it is easy to make mistakes, and formulate a set of discrete equations that will not converge. It is very easy to specify logical conditions that are contradicting themselves. In the Petri–net implementation, this leads to oscillations of discrete state variables with infinite frequency, i.e., it prevents the algorithm from finding a consistent initial state, from which the continuous simulation can be started.

For example, the discrete "equation":

$$p_1 = \textbf{not } \mathrm{pre}(p_1) \tag{9.63}$$

should not be contained in the set of discrete equations, as this will lead to an oscillation between the two states *true* and *false* that will never end. If we mean to toggle between two discrete states as a response to a state–event being triggered (a fairly common situation), we need to model this using two separate places with transitions back and forth that get fired by zero–crossing functions.

How can complex zero–crossing functions be reduced to simple ones? *or*–conditions can be mapped to a set of parallel transitions located between the same two places. They can thus be easily implemented. *and*–conditions are harder to implement, as they would require transitions to be placed in series with each other. Unfortunately, this cannot be done without introducing a new place between them. Thus, *and*–conditions invariably call for an increase in the number of discrete states.

We shall demonstrate this concept by means of the DFSM of Fig. 9.40. We recognize that we wouldn't need the *and*–conditions on the zero–crossing functions in this example, if we were to have available separate discrete states called *StartForward* and *StartBackward*. Thus, we shall decompose the state *Slipping* again into three separate discrete states. Luckily, we know the conditions for switching between them.

We don't need to draw the modified Petri net, as is looks *exactly* like the one of Fig. 9.43. Only the interpretation of the zero–crossing functions is now different. They are:

$$c_1 = v < 0 \tag{9.64a}$$
$$c_2 = v > 0 \tag{9.64b}$$
$$c_3 = v == 0 \tag{9.64c}$$
$$c_4 = v < 0 \tag{9.64d}$$
$$c_5 = v > 0 \tag{9.64e}$$
$$c_6 = \hat{s} < -1 \tag{9.64f}$$

$$c_7 = \hat{s} > 1 \tag{9.64g}$$
$$c_8 = \hat{s} >= -1 \tag{9.64h}$$
$$c_9 = \hat{s} <= 1 \tag{9.64i}$$
$$c_10 = v >= 0 \tag{9.64j}$$
$$c_11 = v <= 0 \tag{9.64k}$$

As expected, all of the zero–crossing functions are now simple functions consisting of a single relational operation only.

9.14 Summary

In this chapter, we have dealt with heavily discontinuous models. We have shown that integration algorithms should be spared from having to deal with discontinuous models directly. Two types of event descriptions were introduced, the time events and the state events, that enable the simulation software to treat discontinuous models in a safe and efficient manner, while protecting the integration algorithms from them. Special root finding algorithms were discussed that are particularly well suited to locate state events.

Event descriptions are quite general, and can be used to deal with most types of discontinuities adequately from a numerical point of view. Exceptions may be the propagation of discontinuous functions through conservation equations. If a step enters an ideal wave equation, a discontinuity will occur that travels through space with time. Consequently, the event times will be infinitely dense, which, from a practical point of view, doesn't make any sense. Adequate handling of discontinuities in hyperbolic PDEs is a very difficult task, and no good answer has been found to date for tackling this challenging problem. The best answer currently available is to apply a variable transformation that will ensure that the waves travel at least along the axes of the coordinate system rather than in an arbitrary direction, which boils down to using the *method of characteristics*. However, even this approach doesn't *solve* the problem. It only alleviates it somewhat.

It was also shown that event descriptions are awkward when dealing with complex engineering models. They are low–level constructs that should not be viewed as modeling elements, but only as intermediate descriptions that are automatically being generated by the model compiler on the way of transforming the model, as specified by the user, into a simulation program that can be executed safely and efficiently using numerical integration software.

Higher–level constructs were introduced in the form of object–oriented *if*–expressions and *when*–clauses, and several fairly advanced applications of these tools have been demonstrated.

It was finally shown that, although the description mechanisms using these constructs are general and convenient, currently available modeling software (i.e., Dymola) is still unable to translate all possible (and physically meaningful) models described using these constructs down into properly executable simulation code. Variable structure models may be contaminated by conditional index changes that require special handling, such as inlining those integrators that are responsible for the partial constraint on a switch equation. Sometimes it is also possible to apply a generalized version of the Pantelides algorithm instead. Whereas it should be possible to at least automate the former approach using the inlining technique, this has not yet been attempted in the current version of the Dymola model compiler.

9.15 References

[9.1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986. 500p.

[9.2] Ilia Nikolaevich Bronshtein and Konstantin Adolfovich Semendiaev. *A Guide–Book to Mathematics*. H. Deutsch Publishing, Frankfurt am Main, Germany, 1971.

[9.3] Michael B. Carver. Efficient Handling of Discontinuities and Time Delays in Ordinary Differential Equation Simulations. In Mohammed H. Hamza, editor, *Proceedings Simulation'77*, pages 153–158, Montreux, Switzerland, 1977. Acta Press.

[9.4] François E. Cellier and Hilding Elmqvist. Automated Formula Manipulation Supports Object–oriented Continuous System Modeling. *IEEE Control Systems*, 13(2):28–38, 1993.

[9.5] François E. Cellier. *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools*. PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1979.

[9.6] John R. Dormand and Peter J. Prince. Runge–Kutta Triples. *J. of Computational and Applied Mathematics*, 12A(9):1007–1017, 1986.

[9.7] John R. Dormand and Peter J. Prince. Runge–Kutta–Nyström Triples. *J. of Computational and Applied Mathematics*, 13(12):937–949, 1987.

[9.8] Steven L. Dvorak, Richard W. Ziolkowski, and Donald G. Dudley. Ultra–Wideband Electromagnetic Pulse Propagation in a Homogeneous Cold Plasma. *Radio Science*, 32(1):239–250, 1997.

[9.9] Edda Eich-Söllner. *Projizierende Mehrschrittverfahren zur numerischen Lösung von Bewegungsgleichungen technischer Mehrkörpersysteme mit Zwangsbedingungen und Unstetigkeiten.* PhD thesis, Universität Augsburg, Augsburg, Germany, 1991.

[9.10] Edda Eich-Söllner. Convergence Results for a Coordinate Projection Method Applied to Mechanical Systems With Algebraic Constraints. *SIAM J. of Numerical Analysis*, 30(5):1467–1482, 1993.

[9.11] Hilding Elmqvist, François E. Cellier, and Martin Otter. Object–Oriented Modeling of Hybrid Systems. In *Proceedings ESS'93, European Simulation Symposium*, pages xxxi–xli, Delft, The Netherlands, 1993.

[9.12] Hilding Elmqvist. *Dymola — Dynamic Modeling Language, User's Manual, Version 5.3.* DynaSim AB, Research Park Ideon, Lund, Sweden., 2004.

[9.13] Gerald Grabner and Andrés Kecskeméthy. Reliable Multibody Collision Detection Using Runge–Kutta Integration Polynomials. In *Proceedings International Conference on Andvances in Computational Multibody Dynamics*, Lisbon, Portugal, 2003.

[9.14] Nicola Guglielmi and Ernst Hairer. Implementing Radau–IIA Methods for Stiff Delay Differential Equations. *Computing*, 67:1–12, 2001.

[9.15] Mary Kathleen Horn. *Developments in High Order Runge–Kutta–Nyström Formulas.* PhD thesis, University of Texas at Austin, 1977.

[9.16] Mary Kathleen Horn. Fourth– and Fifth–Order Scaled Runge–Kutta Algorithms for Treating Dense Output. *SIAM J. of Numerical Analysis*, 20:558–568, 1983.

[9.17] Katsushi Ito and Makiko Nisio. On Stationary Solutions of a Stochastic Differential Equation. *J. of Mathematics of Kyoto University*, 4(1):1–75, 1964.

[9.18] Matthias Krebs. Modeling of Conditional Index Changes. Master's thesis, Dept. of Electrical & Computer Engineering, University of Arizona, Tucson, Ariz., 1997.

[9.19] Shengtai Li and Linda R. Petzold. Moving Mesh Methods with Upwinding Schemes for Time–Dependent PDEs. *J. of Computational Physics*, 131:368–377, 1997.

[9.20] Pieter J. Mosterman, Martin Otter, and Hilding Elmqvist. Modeling Petri Nets As Local Constraint Equations For Hybrid Systems Using Modelica. In *Proceedings SCSC'98, Summer Computer Simulation Conference*, pages 314–319, Reno, Nevada, 1998.

[9.21] Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[9.22] Martin Otter, Hilding Elmqvist, and Sven Erik Mattsson. Hybrid Modeling in Modelica Based On Synchronous Data Flow Principle. In *Proceedings IEEE, International Symposium on Computer Aided Control System Design*, pages 151–157, Kohala Coast, Hawaii, 1999.

[9.23] Constantinos Pantelides. The Consistent Initialization of of Differential–Algebraic Systems. *SIAM Journal of Scientific and Statistical Computing*, 9(2):213–231, 1988.

[9.24] Anton Schiela and Hans Olsson. Mixed–mode Integration for Real–time Simulation. In *Proceedings Modelica'2000 Workshop*, pages 69–75, Lund, Sweden, 2000.

[9.25] Hans Schlunegger. *Untersuchung eines netzrückwirkungsarmen, zwangskommutierten Triebfahrzeugstromrichters zur Einspeisung eines Gleichstromzwischenkreises aus dem Einphasennetz.* PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1977.

[9.26] Lawrence F. Shampine, Ian Gladwell, and Richard W. Brankin. Reliable Solutions of Special Event Location Problems for ODEs. *ACM Transactions on Mathematical Software*, 17(1):11–25, 1991.

9.16 Bibliography

[B9.1] Brian Armstrong-Hélouvry. *Control of Machines With Friction.* Kluwer Academic Publishers, Boston, Mass., 1991.

[B9.2] Carlos A. Canudas de Wit, Hans Olsson, Karl Johan Åström, and Pablo Lischinsky. A New Model for Control of Systems With Friction. In *Proceedings International Conference on Control Theory and Its Applications*, pages 225–229, Kibbutz Maab Hachamisha, Israel, 1993.

[B9.3] René David and Hassane Alla. *Petri Nets and Grafcet.* Prentice–Hall, Upper Saddle River, N.J., 1992.

[B9.4] Martin Otter, Hilding Elmqvist, and François E. Cellier. Modeling of Multibody Systems With the Object–Oriented Modeling Language Dymola. *Journal of Nonlinear Dynamics*, 9(1):91–112, 1996.

[B9.5] Martin Otter. *Objektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter.* PhD thesis, Dept. of Mech. Engr., Ruhr–University Bochum, Germany, 1994.

[B9.6] Friedrich Pfeiffer and Christoph Glocker. *Multibody Dynamics With Unilateral Contacts*. John Wiley & Sons, New York, N.Y., 1996. 318p.

[B9.7] Muhammad H. Rashid. *Spice for Power Electronics and Electric Power*. Prentice–Hall, Englewood Cliffs, N.J., 1994.

[B9.8] Jiri Vlach and Kishore Singhal. *Computer Methods for Circuit Analysis and Design*. Van Nostrand Reinhold, New York, second edition, 1994.

9.17 Homework Problems

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

Implement in MATLAB the RKF4/5 algorithm introduced in Chapter 3 of this book together with the optimistic step–size control algorithm of Eq.(3.89).

Add a root solver (RKF4/5RT) to the method that is based on an implementation of the *Regula Falsi* algorithm.

[H9.2] Runge–Kutta–Fehlberg with Root Solver

Repeat Hw.[H9.1]. This time around, we wish to add a root solver based on an implementation of the *Golden Section* algorithm to the method.

[H9.3] Runge–Kutta–Fehlberg with Root Solver

Repeat Hw.[H9.1]. This time around, we wish to add a root solver based on an implementation of *direct cubic interpolation* to the method.

[H9.4] Direct Hermite Interpolation

We wish to improve the solution to Hw.[H9.3]. Rather than solving for the coefficients of the cubic interpolation polynomial directly using matrix inversion, we want to define a set of spanning polynomials, similar to the way introduced earlier in the chapter in the implementation of the *inverse Hermite interpolation* algorithm.

[H9.5] The Mechanical Loose Element

The functioning of a mechanical loose element is illustrated graphically in Fig.H9.5a.

The output, y, lags behind the input, x, by no more than the distance, d. If the direction of x changes, y remains constant, until it again lags behind by d, now in the opposite direction.

Model the loose element using *if*– and *when*–statements such that the model equations can be sorted appropriately.

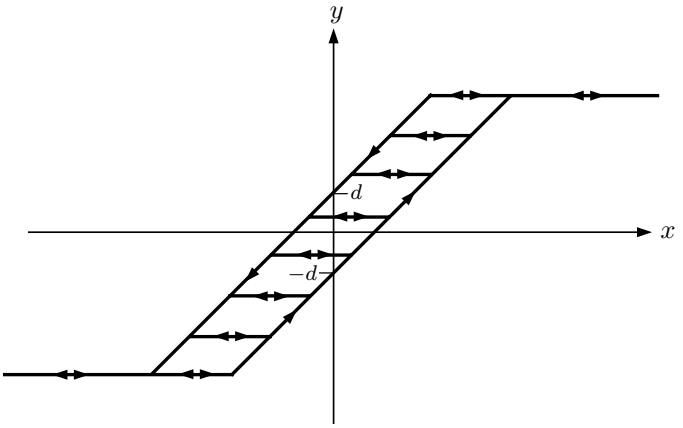

FIGURE H9.5a. Mechanical loose element.

[H9.6] Quantization With Hysteresis

The hysteretic quantization function is illustrated graphically in Fig.H9.6a.

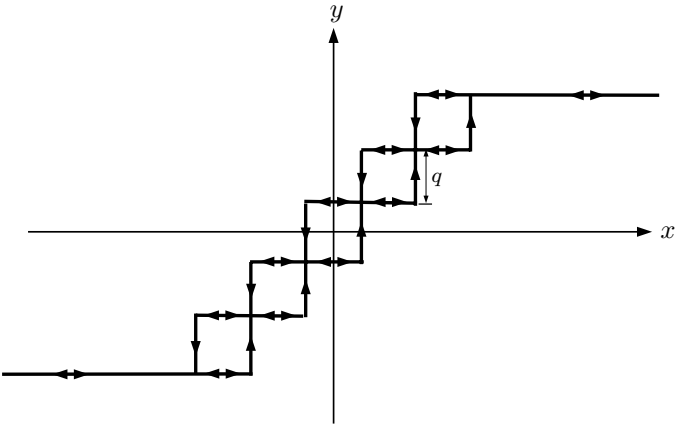

FIGURE H9.6a. Hysteretic quantization function.

The output, y, stays always in the vicinity of the input, x. The distance between them is never greater than half of the quantization distance, $q/2$. Yet, whereas x can change continuously over time, y is a discrete state variable.

Model the hysteretic quantization element using *if*– and *when*–statements such that the model equations can be sorted appropriately.

[H9.7] Thyristor

We wish to model the thyristor described earlier in the chapter by means of *if*–statements. The thyristor element is depicted in Fig. H9.7a.

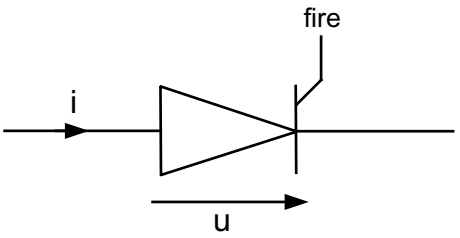

FIGURE H9.7a. Thyrisor.

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.

Convert all *if*–statements of the thyristor model to their algebraic equivalents. Write down all of the equations governing the thyristor–controlled rectifier circuit of Fig. H9.7b.

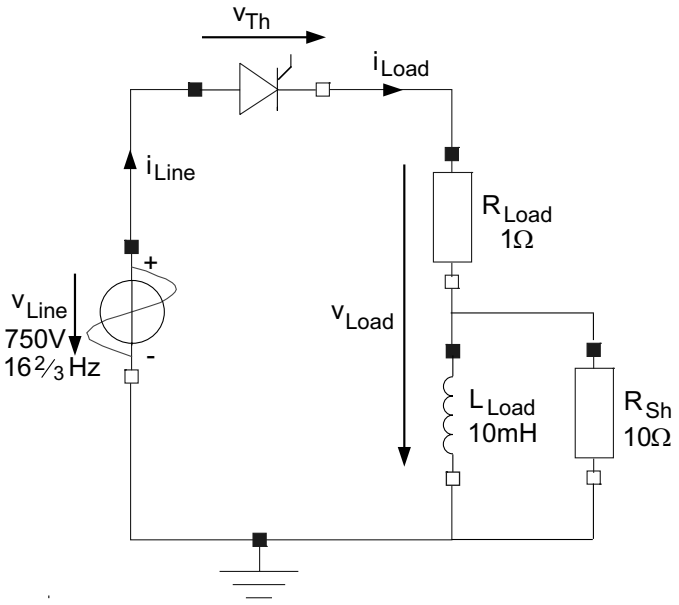

FIGURE H9.7b. Thyrisor–controlled rectifier circuit.

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 hori-

zontally and vertically using the variable substitution technique.

Using any one of the integration algorithms of Hw.[H9.1–4], simulate the model in MATLAB across 0.2 seconds. 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_{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.

[H9.8] Thyristor

We wish to repeat the simulation of Hw.[H9.7] for the modified thyristor–controlled rectifier circuit of Fig. H9.8a.

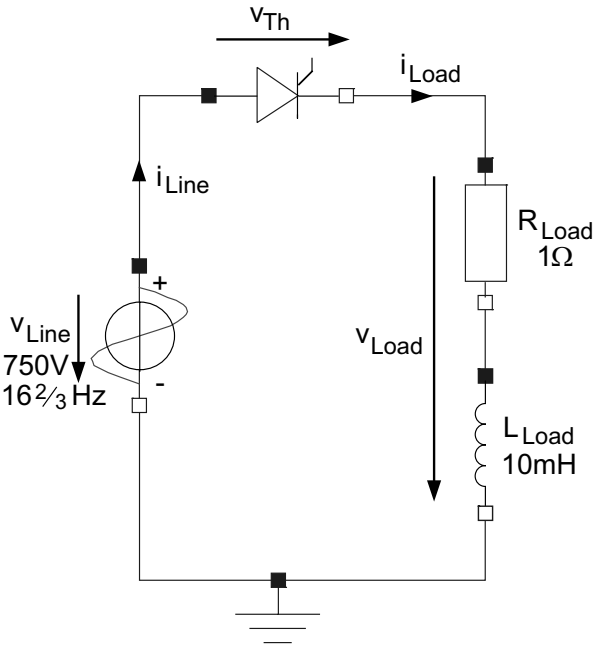

FIGURE H9.8a. Thyrisor–controlled rectifier circuit.

Draw the structure digraph of the resulting equation system, and show that the switch equations do not appear inside an algebraic loop.

Inline the integrator for the inductor using backward Euler. Draw the structure digraph of the modified equation system. Show that the switch equations now indeed appear inside an algebraic loop.

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

Simulate the model in MATLAB across 0.2 seconds. 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_{Line}. During all other times, it is set to *false*. Since there is no integrator left in the model, you cannot use RKF4/5RT any longer. Instead, you need to program the iteration on the zero–crossing function directly into the simulation program. Plot the load voltage, v_{Load}, as well as the load current, i_{Load}, as functions of time.

[H9.9] Zener Diode

No diode can hold current against an arbitrarily strong electrical field. Thus, if the negative voltage across the diode becomes too large, we are confronted with *avalanche breakdown*. The diode suddenly starts conducting again.

A Zener diode makes use of the avalanche breakdown phenomenon, by constructing a diode such that avalanche breakdown occurs early and at a well defined voltage.

Zener diodes are not used like regular diodes, but rather as reverse diodes. Thus, the voltage, in a Zener diode, is defined positive from the cathode to the anode, rather than from the anode to the cathode.

Figure H9.9a shows the Zener diode element together with its voltage and current conventions.

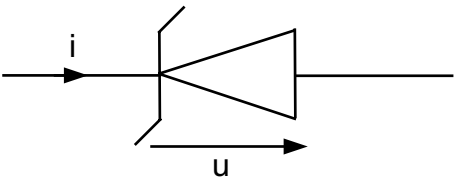

FIGURE H9.9a. Zener diode.

The current/voltage characteristic of the ideal Zener diode is shown in Fig. H9.9b.

The voltage u_B is the *breakdown voltage* of the device.

Zener diodes are commonly placed in parallel with delicate equipment, such as electro–motors. Their purpose is to protect the equipment from potential damage caused by high voltage.

Use the parameterized curve description technique to derive a model of the ideal Zener diode.

[H9.10] Tunnel Diode

A tunnel diode is a regular diode with a tunnelling effect in the conducting area of the device. The tunnel diode element is shown in Fig. H9.10a together with its voltage and current conventions.

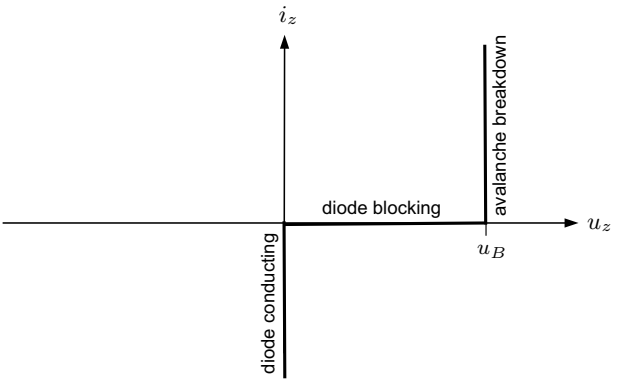

FIGURE H9.9b. Ideal Zener diode characteristic.

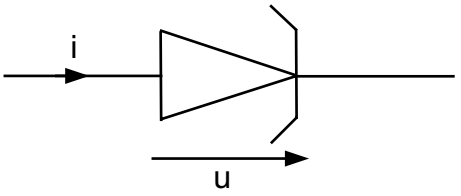

FIGURE H9.10a. Tunnel diode.

The current/voltage characteristic of a typical tunnel diode are shown in Fig. H9.10b.

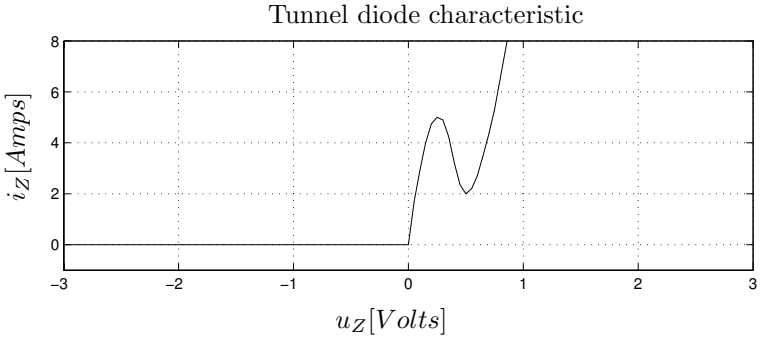

FIGURE H9.10b. Typical tunnel diode characteristic.

When the voltage across the tunnel diode becomes positive, the tunnel diode, just like a regular diode, starts conducting. Yet, the current doesn't grow as rapidly as in the case of a regular diode. With increasing voltage, the current first starts growing, then it decays once more (the tunnelling

effect), before it starts growing rapidly like with a regular diode.

Tunnel diodes are sometimes used for constructing nonlinear oscillator circuits.

We wish to idealize the tunnel diode. To this end, we shall describe it by the idealized characteristic of Fig. H9.10c.

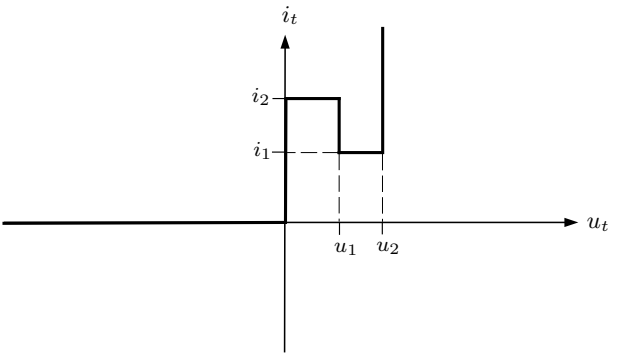

FIGURE H9.10c. Ideal tunnel diode characteristic.

Derive a model of the ideal tunnel diode using the parameterized curve description technique.

[H9.11] Friction

Translate the DFSM of Fig. 9.40 into a set of Boolean expressions governing the four states and their transitions.

Integrate this model with the model of the simplified friction characteristic of Fig. 9.39 developed in the chapter, and convince yourself by manual simulation that the integrated model represents the simplified friction characteristic correctly under all operating conditions.

[H9.12] Dry Hysteresis

Given the dry hysteresis function of Fig. 9.22. Let us assume that $x_1 = y_1 = -1$, and $x_2 = y_2 = +1$. We wish to drive that model using the input:

$$x(t) = 2 \cdot \cos(t) \tag{H9.12a}$$

Derive a Petri net description of the dry hysteresis function. Develop generic synchronous data flow models for the different types of places and transitions encountered in the model.

Extract all of the equations of the discrete model as well as the zero–crossing functions. Implement the model in MATLAB using a suitable algorithm for state–event detection.

Simulate the model in MATLAB across 10 seconds of simulated time, and plot y as a function of x.

[H9.13] Limiter Function

Given the limiter function of Fig. H9.13a.

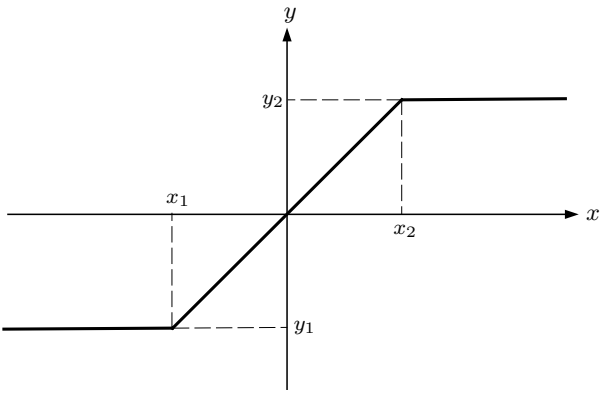

FIGURE H9.13a. Limiter function.

Let us assume that $x_1 = y_1 = -1$, and $x_2 = y_2 = +1$. We wish to drive that model using the input:

$$x(t) = 2 \cdot \cos(t) \tag{H9.13a}$$

Derive a Petri net description of the limiter function. Develop generic synchronous data flow models for the different types of places and transitions encountered in the model.

Extract all of the equations of the discrete model as well as the zero–crossing functions. Implement the model in MATLAB using a suitable algorithm for state–event detection.

Simulate the model in MATLAB across 10 seconds of simulated time, and plot y as a function of x.

9.18 Projects

[P9.1] State Event Localization

In this chapter, we talked little about the use of linear multi–step methods in the simulation of discontinuous models. The reason is that the overhead associated with restarting such a method after an event has occurred is too large to make these methods attractive for the simulation of models containing frequent discontinuities.

Yet, multi–step techniques have an advantage over single–step algorithms due to the availability of the Nordsieck vector. The Nordsieck vector makes it possible to find the zero crossing of a zero–crossing function expressed

as a state variable through *interpolation* instead of *iteration*. This can be done using an interpolation polynomial of the same order of approximation accuracy as the integration method itself. Therefore, zero crossings found in this way are almost as accurate as those found by iteration. They may still be a little less accurate, because the iteration technique involves a reduction of the integration step size in the vicinity of the event, whereas the interpolation method does not.

We had to use iteration in the case of the Runge–Kutta algorithms, because the solution is only available to us with full approximation accuracy at the end of the interval, not at any point in between.

The problem of finding interpolation algorithms for Runge–Kutta methods was first tackled by Horn [9.15, 9.16]. The most commonly used codes today offering implementations of explicit Runge–Kutta algorithms with *dense output* interpolation algorithms are codes based on DOPRI4/5 [9.6, 9.7].

The DOPRI4/5 algorithm is characterized by the Butcher tableau:

0	0	0	0	0	0	0	0
$\frac{1}{5}$	$\frac{1}{5}$	0	0	0	0	0	0
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$	0	0	0	0	0
$\frac{4}{5}$	$\frac{44}{45}$	$\frac{-56}{15}$	$\frac{32}{9}$	0	0	0	0
$\frac{8}{9}$	$\frac{19372}{6561}$	$\frac{-25360}{2187}$	$\frac{64448}{6561}$	$\frac{-212}{729}$	0	0	0
1	$\frac{9017}{3168}$	$\frac{-355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$\frac{-5103}{18656}$	0	0
1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$\frac{-2187}{6784}$	$\frac{11}{84}$	0
x_1	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$\frac{-92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$
x_2	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$\frac{-2187}{6784}$	$\frac{11}{84}$	0

where:

$$f_1(q) = 1 + q + \frac{1}{2}q^2 + \frac{1}{6}q^3 + \frac{1}{24}q^4 + \frac{1097}{120000}q^5 + \frac{161}{120000}q^6 + \frac{1}{24000}q^7$$

$$f_2(q) = 1 + q + \frac{1}{2}q^2 + \frac{1}{6}q^3 + \frac{1}{24}q^4 + \frac{1}{120}q^5 + \frac{1}{600}q^6$$

In DOPRI4/5, usually the 5^{th}–order accurate algorithm is propagated,

whereas the 4^{th}–order accurate algorithm is used for step–size control purposes.

Dormand and Prince determined that a third algorithm can be added without adding an additional stage:

$$x_3(\sigma) = x_n + \sigma \cdot h \cdot \sum_{i=1}^{7} \hat{b}_i(\sigma) \cdot f_i \tag{P9.1b}$$

where:

$$\sigma \in [0, 1]$$

The third approximation polynomial, $x_3(\sigma)$, is parameterized in an additional parameter σ. It offers a 5^{th}–order accurate smooth interpolation polynomial valid anywhere between t_n and t_{n+1}, where σ denotes the percentage of the step taken, i.e.

$$x(\sigma) = x(t_\sigma) = x(t_n + \sigma \cdot h) \tag{P9.1c}$$

Thus:

$$x_3(\sigma = 0) = x_n \tag{P9.1d}$$
$$x_3(\sigma = 1) = x_2 = x_{n+1} \tag{P9.1e}$$

The coefficients $\hat{b}_i$ are cubic polynomials in σ[9.13]:

$$\hat{b}_1 = -\frac{435\sigma^3 - 1184\sigma^2 + 1098\sigma - 384}{384} \tag{P9.1f}$$
$$\hat{b}_2 = 0 \tag{P9.1g}$$
$$\hat{b}_3 = \frac{500\sigma(6\sigma^2 - 14\sigma + 9)}{1113} \tag{P9.1h}$$
$$\hat{b}_4 = -\frac{125\sigma(9\sigma^2 - 16\sigma + 6)}{192} \tag{P9.1i}$$
$$\hat{b}_5 = \frac{729\sigma(35\sigma^2 - 64\sigma + 26)}{6784} \tag{P9.1j}$$
$$\hat{b}_6 = -\frac{11\sigma(3\sigma - 2)(5\sigma - 6)}{84} \tag{P9.1k}$$
$$\hat{b}_7 = \frac{\sigma(\sigma - 1)(5\sigma - 3)}{2} \tag{P9.1l}$$

Dense output interpolation was originally designed as a means to facilitating the display of smoother output curves. Yet, the technique is very useful for the localization of zero–crossing functions in discontinuous models as well. Other applications concern the simulation of *delay–differential equations*, and also aspects of *real–time simulation*, as we shall demonstrate in the next chapter of this book.

Assuming that the derivatives of all zero–crossing functions have been added to the model as additional state equations, the zero–crossing functions themselves are state variables, for which dense interpolation is available.

Assuming further that $x_n \cdot x_{n+1} < 0$ for any of the zero–crossing states, we can find the corresponding next event time t_{next} by computing the value $\hat{\sigma}$, for which $x_3(\hat{\sigma}) = 0$. Then, $t_{\text{next}} = t_{\hat{\sigma}} = t_n + \hat{\sigma} \cdot h$.

Develop effective algorithms for determining $\hat{\sigma}$, and compare the computational efficiency of the interpolation technique with that of the earlier introduced iteration techniques.

[P9.2] State Event Detection

We have demonstrated in this chapter that state events may be missed, if the corresponding zero–crossing functions exhibit two zero crossings that are only separated by a short distance in time.

One approach to dealing with this problem, as demonstrated in this chapter, is through adding *unimportant state events* to the set of events to be iterated upon by appending the derivative of the original zero–crossing function as an additional zero–crossing function to the set.

Yet, this is not the only way of tackling this problem. Another approach has been described in the literature that might be worth considering as an alternative.

Given an n^{th}–order polynomial:

$$p_0(t) = t^n + a_{n-1} \cdot t^{n-1} + a_{n-2} \cdot t^{n-2} + \cdots + a_1 \cdot t + a_0 \tag{P9.2a}$$

We can define the following series of polynomials:

$$p_1(t) = \frac{d}{dt} p_0(t) \tag{P9.2b}$$

and:

$$p_2(t) = -\text{rem}\left(\frac{p_0(t)}{p_1(t)}\right) \tag{P9.2c}$$

$$\vdots \tag{P9.2d}$$

$$p_m(t) = -\text{rem}\left(\frac{p_{m-2}(t)}{p_{m-1}(t)}\right) \tag{P9.2e}$$

where the *rem*–operator denotes the remainder of the polynomial division. Such a series of polynomials is called a *Sturm sequence* [9.2].

If we wish to determine, how many zero crossings the polynomial $p_0(t)$ has in the time interval $[t_a, t_b]$, we can evaluate the polynomials of the Sturm sequence for $t = t_a$ and for $t = t_b$. We count the number of sign changes in the values of the Sturm sequence separately at both ends. The

difference between the number of sign changes at both ends equals the number of zero crossings of the polynomial $p_0(t)$ in the interval $[t_a, t_b]$.

If the zero–crossing function has been defined as a state variable, and if we simulate the model using a Runge–Kutta triple, we have an n^{th}–order interpolation polynomial available, as was shown in Pr.[P9.1].

Thus, we can define the Storm sequence of that interpolation polynomial and determine accurately, how many zero crossings occur within the time interval $[t_n, t_{n+1}]$ [9.26].

Study, how the Sturm sequence can be implemented most effectively.

Compare algorithms for detection of *short–living state events* that are based on augmented sets of zero–crossing functions with methods based on the Sturm sequence for their computational efficiency and reliability.

[P9.3] Delay–Differential Equations

Delay–differential equations are frequently encountered in geological engineering applications and also in chemical process engineering models. In these types of applications, it happens frequently that one process generates some material that is then transported to another process, where it is being used as an input. Other applications of delay–differential equations include the remote control of equipment in space, where the communication delays have to be taken into account.

In all of these cases, we encounter delay–differential equations of the form:

$$\dot{x}_2(t) = f(x_1(t - \Delta)) \tag{P9.3a}$$

The problem here is that the time instant $t - \Delta$ may not be an output point, or even the end of an integration step.

In many applications, a small error in the delay, Δ does not matter. However when it does matter, i.e., if there is a feedback loop back from x_2 to x_1, then we have a problem.

If the model is simulated using a linear multi–step algorithm, it no longer suffices to store the state variables at each output point. We need to store the entire Nordsieck vector at the end of each integration step for at least Δ time units, so that we can appropriately interpolate to evaluate x_1 at time $t - \Delta$.

If the model is simulated using a single–step algorithm, it may again be preferable to use one of the Runge–Kutta triples. However in that case, we would need to store the solution of every stage of the algorithm at the end of each integration step for at least Δ time units, so that we can appropriately interpolate to evaluate x_1 at time $t - \Delta$.

Although this technique doesn't create any principle difficulties, it causes significant computational overhead. The issue thus is how solvers for delay–differential equations can be implemented in a computationally efficient way. *Circular shift registers* are one approach that comes to mind, but this may not be the only one, or even the best approach to dealing with this

problem.

Study computationally efficient ways of data storage and retrieval for the numerical simulation of delay–differential equations, and modify existing codes to implement those.

9.19 Research

[R9.1] Stiff Discontinuous Models

If a discontinuous model is stiff, we must use an implicit integration algorithm to simulate it. Although we could use a code, such as DASSLRT, this may be quite inefficient, because linear multi–step methods are hardly ever suitable for dealing with heavily discontinuous models due to the overhead and inaccuracy associated with the start–up algorithm needed after each event.

Thus, it is important to extend the idea of an interpolation polynomial to obtain dense output from the explicit Runge–Kutta algorithms to implicit ones, such as the Radau–IIA, or Lobatto–IIIC algorithms introduced earlier in this book.

The problem has been recognized, and a number of research groups are currently working on this issue. First results have recently been published [9.14].

Yet, the problem is still essentially unsolved. The reason is that the interpolated result needs to be propagated to the next step. Thus, it is insufficient to prove that the interpolated result is n^{th}–order accurate. We ought to prove in addition that it is also numerically stable.

[R9.2] Discontinuous Hyperbolic PDEs

Whereas we have discussed in this chapter the problems associated with the detection and localization of state events, we always made the assumption that the event times are somewhat spaced out, i.e., within a finite time interval, the number of events must remain finite.

Unfortunately, this assumption does not always hold true. If we apply a discontinuity to a hyperbolic PDE, such as the wave equation, the discontinuity travels through the medium with time, i.e., at any point in time, the discontinuity can be located somewhere in the medium. Hence the event times are no longer spread out.

Some researchers have applied moving grid methods to these types of problems [9.19]. Others have applied frequency–domain techniques [9.8]. Yet, whereas these techniques may be suitable to track steep wave fronts, neither of these techniques is geared to dealing with true discontinuities.

How do we know that inaccuracies in estimating, where the discontinuity is located at any point in time will not propagate through the solution and accumulate as time passes? Do we have any handle on the numerical

stability problems associated with these types of situations?

There must exist better ways to calculate with arbitrary accuracy, where the discontinuity is located when, and tackle the problem by subdividing the domain into "left" and "right" regions in space, and "before" and "after" domains in time, and extrapolate (interpolate) to the location of the discontinuity from all sides.

[R9.3] Sliding Motion

Sliding motion is a second type of problem that can lead to events with infinite frequency of occurrence.

In this chapter, we have encountered creeping behavior of a simulation code implementing a discontinuous model twice.

The first time was in the context of the train engine model. However, the creeping behavior only occurred because we had implemented the discontinuity handling incorrectly. Once we solved that problem, the creeping behavior went away.

The second time, we ran into a similar problem was in the context of one of our friction models, where we found that coming out of sticking friction caused the model to be thrown back into sticking friction immediately again. This happened, in spite of the fact that the model is formally correct.

Here, we were able to tackle the problem by introducing two additional discrete states, *StartForward* and *StartBackward*. Once these states had been introduced and the state transition logic had been undated appropriately, the problem went away.

Is this the worst that can happen? Unfortunately, the answer to this question is negative. Let us explain this assertion by means of an example.

Figure R9.3a shows a flying vehicle on a slow collision course with a sloped wall.

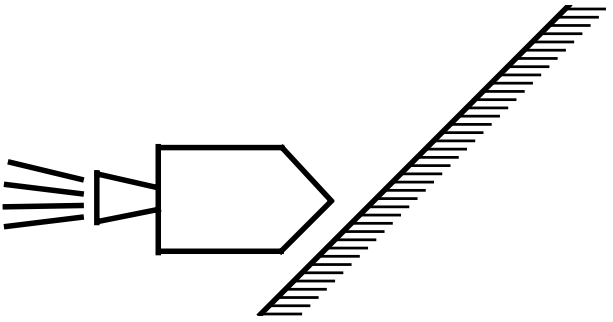

FIGURE R9.3a. Sliding motion.

Once the vehicle arrives at the wall, it either gets stuck there, or if the thrust is sufficiently large to overcome sticking friction, it will glide up the slope, as it has no choice in the matter.

Unfortunately, it is a rather difficult problem to convince the simulation code that this is what must happen. If Newton's law is being formulated separately for the horizontal and vertical motions, the vehicle cannot move forward at all, as the wall is in the way. It can only move upward. However as it moves upward, it no longer remains in contact with the wall. Thus, the vehicle starts moving forward again. However by doing so, it bumps immediately back into the wall. The model ends up with state events occurring at infinite frequency. Of course, the problem will go away, if we modify the coordinate system to coincide with the slope of the ramp.

Although the example looks somewhat academic, the problem itself is quite realistic, and these types of problems indeed occur frequently in mechanical systems with closed kinematic loops, such as the simulation of a car moving on a road. As all four wheels are in contact with the ground, we are faced with multiple closed kinematic loops. If the car drives around a bend, two of the wheels need to move a little faster than the other two. Any numerical discrepancy between the simulated motion and the physical constraint will invariably lead to the type of behavior explained above.

These types of problems have been studied in recent years [9.9, 9.10]. Yet, no fully automated algorithms have been designed that can detect these problems and modify the problem formulation automatically and on the fly in such a way as to remove the events occurring with infinite frequency.

[R9.4] Simulation of Noisy Models

A third type of problems that will lead, in the theoretical limit, to a series of events occurring with infinite frequency is the simulation of models with noise.

Most continuous–system simulation software offers at least uniform and Gaussian distributed random number generators that enable the modeler to superpose noise to some input signals of his or her model. The noise signal may e.g. be used to describe the headwind facing a helicopter in flight, or it may be used to describe the unevenness of a road along which a vehicle is driving. In the case of the helicopter, the purpose of including the headwind may be to test the robustness of the control algorithm. In the case of the road vehicle, it may be to simulate the behavior of the shock absorbers.

Unfortunately, the random number generator is a rather dubious modeling element, as it changes its behavior as a function of the integration step size used.

Uncorrelated white noise ought to have a frequency spectrum that is totally flat at all frequencies. Yet, plotting the frequency spectrum of a random number generator used in a simulation model, we notice that the spectrum eventually decays as $1/f$. The bandwidth of the random number generator is band–limited by the sampling rate. The smaller we choose the step size, the larger the bandwidth of the random number generator will

become.

Although some highly theoretical investigations have looked at the analytical solutions of stochastic differential equations [9.17], this is not useful for our purpose.

The problem that we are confronted with is that we cannot use event handling mechanisms to deal with random signals. Yet, if we ignore them, they will invariably get entangled with the step–size control of the variable–step integration algorithms.

Very little research has been done to date that looks at this problem from a practical perspective.