10

Real–time Simulation

Preview

In this chapter, we shall discuss the special requirements of real–time simulation, i.e., of simulation runs that keep abreast of the passing of real time, and that can accommodate driving functions (input signals) that are generated outside the computer and that are read in by means of analog to digital (A/D) converters.

Until now, computing speed has always been a soft constraint — slow simulation meant expensive simulation, but now, it becomes a very hard constraint. Simulation becomes a *race against time*. If we cannot complete the computations associated with one integration step *before* the real–time clock has advanced by h time units, where h is the current step size of the integration algorithm, the simulation is out of sync, and we just lost the race.

Until now, we always tried to make simulation more comfortable for the user. For example, we introduced step–size controlled algorithms so that the user wouldn't have to worry any more about whether or not the numerical integration meets his or her accuracy requirements. The algorithm would do so on its own. In the context of real–time simulation, we may not be able to afford all this comfort any longer. We may have to throw many of the more advanced features of simulation over board in the interest of saving time, but of course, this means that we have to understand even better ourselves how simulation works in reality.

10.1 Introduction

Several very important applications of simulation require real-time performance.

A *flight simulator* for training purposes is useless if it cannot produce a reflection of the performance of the real aircraft or helicopter or space craft in real time. The trainee uses the simulator because learning often is synonymous with making mistakes ... and mistakes may be too costly when working with the real system.

Model Reference Adaptive Controllers (MRACs) make use of a model of an idealized plant, the reference plant, trying to make the real plant behave as similar as possible to the reference plant [10.34]. However, this requires that the reference plant model be simulated in real time in parallel with the real plant, both being driven simultaneously by the same input signals.

A *watchdog monitor* [10.3, 10.2, 10.47] of a nuclear power station reasons about the sanity of the plant. It has some knowledge of how the plant is supposed to operate, and looks out for significant discrepancies between expected and observed plant behavior. To this end, the watchdog monitor maintains a model of the power plant that it runs in parallel with the real plant, comparing its outputs to the measurement data extracted from the real plant. The watchdog monitor thus contains a real–time simulation of a model of the correctly working power plant. Once it discovers a significant aberration in real plant behavior, it kicks off a *fault discriminator* program that, again in real time, tries to narrow down the source of the fault, i.e., seeks to determine, which of the subsystems of the real plant is malfunctioning. It maintains real-time simulations of abstractions of models of all subsystems that permit it to localize errors to a particular subsystem. Once this has been accomplished, a *fault isolation* program is kicked off that invokes a real–time simulation of a more refined model of the faulty subsystem including models of faulty behavior with the aim of identifying the kind of error that is most likely to have occurred within the faulty subsystem [10.11, 10.12, 10.45].

Conceptually, the implementation of real–time simulation software is straightforward. It contains only four new components:

1. The *real–time clock* is responsible for the synchronization of real time and simulated time. The real–time clock is programmed to send a trigger impulse once every h time units of real time, where h is the current step size of the integration algorithm, and the simulation program is equipped with a busy waiting mechanism that is launched as soon as all computations associated with the current step have been completed, and that checks for arrival of the next trigger signal. The new step will not begin until the trigger signal has been received.

2. The *analog to digital (A/D) converters* are read at the beginning of each integration step to update the values of all external driving functions. This corresponds effectively to a sample and hold (S/H) mechanism. The inputs are updated once at the beginning of every integration step and are then kept constant during the entire step.

3. The *digital to analog (D/A) converters* are set at the end of each integration step, i.e., the newest output information is put out through the D/A converters for inspection by the user, or for driving real hardware (for so–called *hardware–in–the–loop* simulations.

4. *External events* are time events that are generated outside the simulation. External events are used for asynchronous communication with the simulation program, e.g. for the modification of parameter values, or for handling asynchronous readout requests, or for communication between several asynchronously running computer programs

either on the same or different computers. External events are usually postponed to the end of the current step and replace a portion of the busy waiting period.

Figure 10.1 illustrates the different tasks that take place during the execution of an integration step.

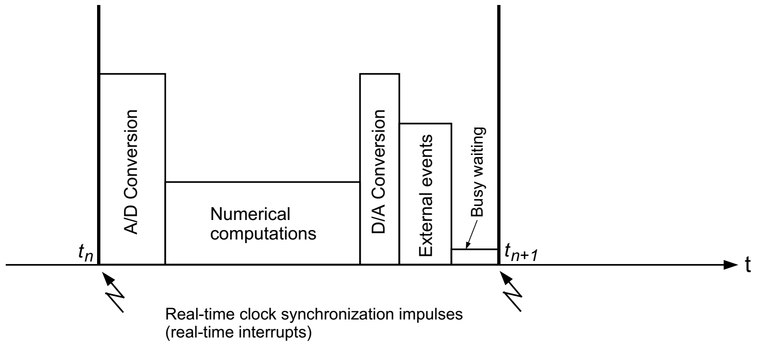

FIGURE 10.1. Task scheduling within integration step.

Once the message from the real–time clock has arrived indicating that the real time has advanced to time t_k, the simulation program first reads all the A/D converters to update the values of all input functions to the current time. It then performs the actual numerical computations associated with the step, calling upon the numerical integration routine and the routine that implements the state–space model. Once this is accomplished, the results are written out to the D/A converters. At this time, the “regular” business associated with the current step are over. The algorithm now consults the “mailbox” in which external events that may have arrived in the meantime are stored, and handles those. Once this has been accomplished, the algorithm has nothing more left to do and enters a “busy waiting” loop in which it repetitively checks the mailbox for arrival of the next message from the real–time clock.

The interprocessor and intertask communication mechanisms can actually be implemented in many different ways. In some cases, it may be desirable to use the waiting time of the processor for background tasks, rather than waste it in a busy waiting loop. In that case, it is not sufficient for the real–time clock to send a message to the simulation program. Instead, it must use the interrupt mechanism of the processor on which the simulation is running to interrupt whatever other task the processor is currently working on.

The difficulties of real–time simulation are *not* of a conceptual nature. They have to do with keeping track of real time. How can we guarantee that all that needs to be accomplished during the integration step can be completed prior to the arrival of the next trigger impulse?

In the previous chapters of this book, we introduced more and more bells and whistles that would help us in being able to guarantee the *correctness* of the simulation results obtained, but all these additional tools were accompanied by some run–time overhead, and in many cases, the amount of time needed to bring these algorithms to completion was not fixed. For example, if we decide to use an implicit integration algorithm, how can we know beforehand how many iterations will be needed to guarantee a prescribed tolerance of the results? However, if we do not limit the number of iterations available to the algorithm, how can we possibly know for sure that the step will be completed before the arrival of the next trigger impulse from the real–time clock? Iteration on state events is a great thing. Yet, can we afford it under real–time conditions? What happens if we do not iterate? Can we still know something about the accuracy of the results obtained? These are the questions that will be discussed in the current chapter.

10.2 The Race Against Time

There are two questions that we can ask ourselves in the context of racing against real time: (i) How can we guarantee that all computations necessary to end the current integration step in time are indeed completed before the next trigger impulse from the real–time clock arrives? (ii) What happens if we don't meet the schedule? Let me first address the second question since it is somewhat easier to deal with.

There are basically four things that we can do if we don't meet the schedule. We can:

1. increase the step size, h, in order to make more time for the tasks that need to be accomplished,
2. make the function evaluation more efficient, i.e., optimize the program that represents our state–space model,
3. improve the speed of the integration algorithm, e.g. by reducing the number of function evaluations necessary during one step, and finally
4. buy ourselves a faster computer.

The last solution may sound like a last resort, but in these times of cheap hardware and expensive software and manpower, it may actually often be the wisest thing to do.

The first solution is interesting. Until now, the step size was always bounded from the top due to accuracy and stability constraints. Now suddenly, the step size is also bounded from the bottom. We cannot reduce the step size to a value smaller than the total real time needed to perform all the computations associated with simulating the model across one step plus the real time needed for dealing with the administration of the simulation during that step. If it happens that the lower bound is larger than the upper, then we are in real trouble.

The second solution is one that has, over the years, been most actively pursued by Granino Korn, who wrote a large number of articles on the issue of how to obtain "cheap" (in the sense of fast) approximations for all kind of functions. He also treated this topic in several of his books [10.26, 10.27].

Many engineering models, such as models used in flight simulators or models of thermal power plants are full of two– and three–dimensional tables representing static characteristics that have been deduced by measurements and for which no explicit formulae are known. The need to interpolate in large three–dimensional tables is a nightmare for designers of real–time simulation software, since these interpolations can be very time consuming, and since the time needed to find the right entries in the table between which to interpolate is not even constant, but depends on the numerical values of the current arguments. Recent advances in neural network technology make it now possible to design feedforward neural networks trained e.g. through accelerated backpropagation algorithms that approximate two– and three–dimensional static functions with arbitrary precision. The training of these networks is slow, but this can be done off–line. Once trained, neural networks are very efficient at run time, providing for very fast multidimensional function evaluation capabilities. Also in this arena, it was Granino Korn who did pioneering work in combining fast neural network technology with high–speed simulation capabilities [10.28].

Finally, the most prominent researchers who dealt (and are still dealing) with the third solution are Jon Smith [10.43] and Bob Howe [10.22, 10.32, 10.33]. Since this approach deals with the numerical integration algorithms themselves, it is most relevant to this textbook, and therefore, we shall talk more about this approach in the current chapter.

Yet, before studying the way of improving the speed of the algorithms, we shall analyze the different methods in order to focus only on those that show suitable features for real–time simulation.

10.3 Suitable Numerical Integration Methods

In real–time simulation, it is not sufficient to obtain a good approximation of the values of the state variables. These approximations are in fact useless, if they arrive too late. We need to make sure that all of the computations associated with a single integration step are completed within the allowed time slot.

To this end, the total number of calculations performed by a single integration step must be bounded, and all of the iterative processes should be cut after a fixed number of iterations. It is evident that this will affect the accuracy of the algorithms, but it is better to obtain a solution with some remaining error, than not be able to obtain it at all within the allowed time [10.18].

Taking into account these considerations, the following analysis tries to examine the different features of the methods introduced in previous chapters of this book in order to discuss their pros and cons in the context of real–time simulation. The analysis is primarily based on Schiela's diploma thesis [10.40].

- *Multi–step methods.* Multi–step methods use information from the previous steps to compute a high–order approximation of the next step. This idea is based on the assumption that the differential equation is smooth, since the approximation uses a polynomial function. Unfortunately, many real–time simulations receive input signals from the real world that are not very smooth. Therefore, multi–step methods may give inaccurate results in such cases.

  On the other hand, multi–step methods reduce the number of function evaluations per step, which is a crucial factor in the real–time context. For this reason alone, and in spite of the fact that some accuracy may be sacrificed in this way, explicit linear multi–step methods, such as Adams–Bashforth, and among them especially those of low order of approximation accuracy, are widely used in real–time simulation [10.23].

- *Explicit single–step methods.* These methods are compatible with the requirements mentioned earlier. Their computational effort is relatively low and constant. The number of calculations per step can be easily estimated. Furthermore, the methods can deal fairly well with discontinuous input signals, since they do not use information from the past. Thus, for non–stiff ordinary differential equations, explicit single–step methods may constitute the best choice.

  However as we already know, these methods have problems with stiff systems. For mildly stiff problems, one remedy is to use integration step sizes that are a fraction of the sample interval, but then the efficiency decays with increasing stiffness.

  A different strategy for stiff systems is to modify the model so that the stiffness decreases. In some cases, the fast dynamics do not significantly influence the overall solution, and under such circumstances, the fast modes can be removed from the model. However, this is not generally the case for stiff systems, and it is always a questionable tactic to change the model in order to get it simulated.

- *Implicit single–step methods.* As we know, implicit methods require solving a system of nonlinear equations at each step, which implies the use of iterative methods, such as Newton iteration. Therefore, the computational effort for each step cannot be estimated reliably, as it depends on the (theoretically unbounded) number of iterations. Hence implicit methods are not suitable for the purpose of real–time simulation. Nevertheless, algorithms based on implicit methods can be used in real–time simulation, provided that the number of iterations is kept bounded to a fixed value.

  However, we must also take into account that, by limiting the number of iterations, we modify the stability domain of these methods.

- *High–order methods.* In most real–time applications, the sampling intervals are small compared to the time scales of interest, and the required accuracy is usually rather low. One important reason for using small sampling intervals is to be able to accommodate real–time input. Input signals must be sampled frequently, since they cannot be reliably interpolated. Taking into account that reducing the amount of calculations at each step is a crucial factor, high–order algorithms will not be suitable for real–time simulation, except under very particular circumstances. It is therefore rare to find real–time simulations that make use of integration methods of orders of approximation accuracy greater than two or three.
- *Variable step methods.* In real–time simulations, we do not have the luxury to be able to change the step size, as it is synchronized with the sampling rate and severely restricted by the real–time specifications of the problem. The best thing that we can do for "controlling" the integration error is to estimate it and log these estimates, so that the quality of the results obtained can at least be judged *a–posteriori.*

  The numerical integration error can be estimated on–line, by comparing the actual simulation with another real–time simulation using a bigger step size. This idea was proposed by Bob Howe [10.23], making use of interpolation techniques to obtain the values of the control run at the sampling times of the actual simulation.

After analyzing all of these features, only *low–order explicit methods* seem well suited for real–time simulation.

In absence of stiffness, discontinuities, or badly nonlinear implicit equations, those methods work properly.

It can happen that the system dynamics are fast compared with the computer clock frequency. In such a case, there is little that we can do except try to optimize the way, in which the calculations are made, or buy ourselves a faster computer.

Leaving high–bandwidth applications aside, real–time simulation of non–stiff smooth systems does not call for any special treatment from a simulation methodology point of view.

Unfortunately, many systems in engineering applications are in fact stiff and, as we already know, explicit methods show poor performance in their integration.

We are unable to solve this problem without using implicit principles but, for the reasons explained above, we must avoid iterative solutions.

These considerations lead us to *semi–implicit* or *linearly implicit* methods and –in a further step– to *multi–rate integration.*

10.4 Linearly Implicit Methods

Linearly implicit or semi–implicit methods exploit the fact that implicit methods applied to linear systems do not require a theoretically unbounded number of iterations. Indeed, the resulting implicit equations can be solved by means of matrix inversion.

A widely used linearly–implicit method is given by the semi–implicit Euler formula [10.40, 10.38]:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot [\mathbf{f}(\mathbf{x_k}, t_k) + \mathcal{J}_{\mathbf{x_k},t_k} \cdot (\mathbf{x_{k+1}} - \mathbf{x_k})] \tag{10.1}$$

where

$$\mathcal{J}_{\mathbf{x_k},t_k} = \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{\mathbf{x_k},t_k} \tag{10.2}$$

is the Jacobian matrix evaluated at $(\mathbf{x_k}, t_k)$.

Notice that

$$\mathcal{J}_{\mathbf{x_k},t_k} \cdot (\mathbf{x_{k+1}} - \mathbf{x_k}) \approx \mathbf{f}(\mathbf{x_{k+1}}, t_{k+1}) - \mathbf{f}(\mathbf{x_k}, t_k) \tag{10.3}$$

and therefore:

$$\mathbf{f}(\mathbf{x_k}, t_k) + \mathcal{J}_{\mathbf{x_k},t_k} \cdot (\mathbf{x_{k+1}} - \mathbf{x_k}) \approx \mathbf{f}(\mathbf{x_{k+1}}, t_{k+1}) \tag{10.4}$$

Thus, the linearly implicit Euler approximates the implicit Euler method. Moreover, in the linear case:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} \tag{10.5}$$

we have:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot [\mathbf{A} \cdot \mathbf{x_k} + \mathcal{J}_{\mathbf{x_k},t_k} \cdot (\mathbf{x_{k+1}} - \mathbf{x_k})] = \mathbf{x_k} + h \cdot \mathbf{A} \cdot \mathbf{x_{k+1}} \tag{10.6}$$

which exactly coincides with Backward Euler. This implies that the stability domain of the linearly implicit Euler method also coincides with the stability domain of Backward Euler.

Equation (10.1) can be rewritten as:

$$(I - h \cdot \mathcal{J}_{\mathbf{x_k},t_k}) \cdot \mathbf{x_{k+1}} = (I - h \cdot \mathcal{J}_{\mathbf{x_k},t_k}) \cdot \mathbf{x_k} + h \cdot \mathbf{f}(\mathbf{x_k}, t_k) \tag{10.7}$$

which shows that $\mathbf{x_{k+1}}$ can be obtained by solving a linear system of equations.

The value of $\mathbf{x_{k+1}}$ can also be obtained as:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot (I - h \cdot \mathcal{J}_{\mathbf{x_k},t_k})^{-1} \cdot \mathbf{f}(\mathbf{x_k}, t_k) \tag{10.8}$$

The formula given by Eq.(10.8) is similar to Forward Euler, but differs in the presence of the term $(I - h \cdot \mathcal{J}_{\mathbf{x_k},t_k})^{-1}$.

¿From a computational point of view, that term implies that the algorithm has to calculate the Jacobian at each step and then either solve a linear equation system or invert a matrix.

Despite the fact that those calculations may turn out to be quite expensive, the computational effort is predictable, which makes the method well suited for real–time simulation.

Taking into account that the stability domain coincides with that of Backward Euler, this method results appropriate for the simulation of stiff and differential algebraic problems. Low–order linearly implicit methods may indeed often be the best choice for real–time simulation. However, they share one drawback with implicit methods: if the size of the problem is large, then the solution of the resulting linear equation system is computationally expensive.

Due to this fact, many different techniques were proposed that optimize how and how often the Jacobian is being evaluated and the linear equation system is being solved [10.19].

We shall not discuss those techniques here for two reasons. First, many of these techniques are designed to make the numerical integration faster *on average*. We are not interested in such approaches in the context of real–time simulation, because we must ensure that the algorithm converges *always* within the allotted time. Second, the statement that the stability domain of the linearly implicit Euler algorithm is the same as that of Backward Euler is only true if the *exact Jacobian* is being used in every step. For example, if we were to approximate the Jacobian by the zero matrix, the method would have the stability domain of Forward Euler.

Many of the implicit algorithms make use of a so–called *modified Newton iteration*. In one variant of that approach, the underlying Hessian matrix is being approximated by a diagonal matrix to make its inversion cheap and painless. This can be done. The price to be paid for this luxury is that the Newton iteration will converge more slowly, i.e., we have to spend more iteration steps, while each individual iteration step is now cheaper. Whether or not this pays off, depends on the application at hand.

Some authors proposed to apply this technique in the case of semi–implicit algorithms as well by approximating the Jacobian through its diagonal elements. We do not recommend this approach. In most cases, this will be the kiss of death, as the stability domain of the method using a diagonal approximation of the Jacobian will most likely loop in the left–half $\lambda \cdot h$ plane, i.e., the method will no longer be stiffly stable.

Other authors proposed to carefully look at the structure of the Jacobian, and at least zero out some of the smallest non–vanishing elements in it. This technique is called *sparsing* [10.40, 10.38], as it makes the Jacobian more sparse, thereby enabling a cheaper linear system solution using either numerical or symbolic sparse matrix techniques. It was shown that sparsing can indeed reduce the computational effort needed to complete the calculations of a single integration step by a significant amount. Yet, the technique must be applied cautiously, as it doesn't take much for the stiffly stable nature of the algorithms to be lost in the process. It thus may generally pay off to work with an accurate computation of the exact Jacobian in each step.

Having said that, we are of course free in how we compute the exact Jacobian, and which technique we use for solving the resulting linear equation system. The Jacobian can either be computed symbolically, leaving it up to the model compiler to find the appropriate expressions by symbolic (algebraic) differentiation, or it can be approximated numerically. Furthermore, Jacobian matrices are usually sparse, because not every state derivative depends on every state variable. Thus, we can use either numerical sparse matrix algorithms in solving the resulting linear system, or we can use one of the two symbolic sparse sparse matrix techniques introduced earlier in this book, i.e., tearing [10.17] or relaxation [10.35].

The improvements achieved by those techniques allow some large stiff systems to be simulated in real time using semi–implicit algorithms. However, there are still larger and more complicated systems, in which these ideas are not enough to win the race against time.

Fortunately, stiffness in large systems is often connected to the presence of some identifiable slow and fast sub–models. In those cases, we can use that information to our advantage by splitting the system and applying different step sizes or even different integration algorithms to the different parts. These ideas lead to the concepts of multi–rate and mixed–mode integration.

Finally, we should mention that several higher–order semi–implicit versions of both multi–step [10.6] and Runge–Kutta [10.1, 10.46] methods have been reported in the literature. We shall not explore these algorithms further, since their principles are similar to those of the linearly implicit Euler method. However, we shall derive one of these methods in a homework problem.

Least suitable among all of the linearly implicit stiffly stable algorithms for the task at hand are those algorithms that are F–stable, in particular the *trapezoidal rule* and its one–legged twin, the *implicit midpoint rule*. The reason for this assertion is the following. Since we cannot use a variable–step algorithm, we are bound to end up with a numerical error in each integration step that is caused by the fixed step size and that is essentially uncontrollable. It is thus recommended to use an algorithm with some additional *artificial damping* to prevent error accumulation [10.18].

10.5 Multi–rate Integration

There are many cases, in which the stiffness is due to the presence of a sub–system with very fast dynamics compared to the rest of the system. Typical examples of this can be found in multi–domain physical systems, since the components of different physical domains usually involve distinct time constants.

For example, if we wish to study the thermal properties of an integrated circuit package, we shall recognize that the electrical time constants of the device are faster in comparison with the thermal time constants by several orders of magnitude. Yet, we cannot ignore the fast time constants, since they are the cause of the heating. In some cases, such as switching power converters, the heating of the device grows with the frequency of switching, i.e., while no switching takes place, the thermal effects are minimal [10.41].

Let us introduce the idea with the following example. Figure 10.2 shows a lumped model of a transmission line fed by a Van–der–Pol oscillator (this example is a variant of an example offered in [10.36]).

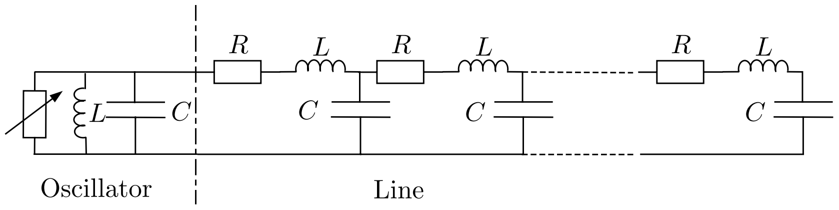

FIGURE 10.2. Van–der–Pol oscillator and transmission line.

We shall assume that the nonlinear resistor of the oscillator circuit satisfies the law:

$$i_R = k \cdot u_R^3 - u_R \tag{10.9}$$

Then the system can be described by the following set of state equations:

$$\frac{di_L}{dt} = \frac{1}{L} u_C \tag{10.10a}$$

$$\frac{du_C}{dt} = \frac{1}{C}(u_C - k \cdot u_C^3 - i_L - i_1) \tag{10.10b}$$

$$\frac{di_1}{dt} = \frac{1}{L} u_C - \frac{R}{L} i_1 - \frac{1}{L} u_1 \tag{10.10c}$$

$$\frac{du_1}{dt} = \frac{1}{C} i_1 - \frac{1}{C} i_2 \tag{10.10d}$$

$$\frac{di_2}{dt} = \frac{1}{L} u_1 - \frac{R}{L} i_2 - \frac{1}{L} u_2 \tag{10.10e}$$

$$\frac{du_2}{dt} = \frac{1}{C} i_2 - \frac{1}{C} i_3 \tag{10.10f}$$

$$\vdots$$

$$\frac{di_n}{dt} = \frac{1}{L}u_{n-1} - \frac{R}{L}i_n - \frac{1}{L}u_n \tag{10.10g}$$
$$\frac{du_n}{dt} = \frac{1}{C}i_n \tag{10.10h}$$

Here, u_C and i_L are the voltage and current of the capacitor and inductance in the oscillator. Similarly, u_j and i_j are the voltage and current of the capacitors and inductances at the j^{th} stage of the transmission line.

Let us assume that the transmission line has 5 stages (i.e., $n = 5$), and the parameters are $L = 10$ mH, $C = 1$ mF, $R = 10\Omega$ and $k = 0.04$.

If we wish to simulate the system using the Forward Euler method, we need to use a step size no greater than $h = 10^{-4}$ seconds. Otherwise, the oscillator output (u_C) is computed with an error that is totally unacceptable.

However, using the input signal generated by the oscillator, the transmission line alone can be simulated with a step size that is 10 times bigger.

Thus, we decided to split the system into two subsystems, the oscillator circuit and the transmission line, using two different step sizes: 10^{-4} seconds for the former, and 10^{-3} seconds for the latter.

In that way, we integrate the fast but small (2nd–order) sub–system using a small step size, whereas we integrate the slow and large (10th–order) sub–system using a larger step size.

As a consequence, during each millisecond of real time, the computer has to evaluate ten times the two scalar functions corresponding to the two first state equations, whereas it only needs to evaluate once the remaining ten functions. Thus, the number of floating–point operations is reduced by about a factor of four compared with a regular simulation using a single step size throughout.

The simulation results are shown in Figs.10.3–10.4.

We can generalize this procedure to systems of the form:

$$\dot{\mathbf{x}}_\mathbf{f}(t) = \mathbf{f}_\mathbf{f}(\mathbf{x}_\mathbf{f}, \mathbf{x}_\mathbf{s}, t) \tag{10.11a}$$
$$\dot{\mathbf{x}}_\mathbf{s}(t) = \mathbf{f}_\mathbf{s}(\mathbf{x}_\mathbf{f}, \mathbf{x}_\mathbf{s}, t) \tag{10.11b}$$

where the sub–indexes, f and s, stand for "fast" and "slow," respectively.

Then, the use of the multi–rate version of Forward Euler with inlining results in a set of difference equations of the form:

$$\mathbf{x}_\mathbf{f}(t_i + (j+1)\cdot h) = \mathbf{x}_\mathbf{f}(t_i + j\cdot h) + h\cdot \mathbf{f}_\mathbf{f}(\mathbf{x}_\mathbf{f}(t_i + j\cdot h), \mathbf{x}_\mathbf{s}(t_i + j\cdot h), t_i + j\cdot h) \tag{10.12a}$$
$$\mathbf{x}_\mathbf{s}(t_i + k\cdot h) = \mathbf{x}_\mathbf{s}(t_i) + h\cdot \mathbf{f}_\mathbf{s}(\mathbf{x}_\mathbf{f}(t_i), \mathbf{x}_\mathbf{s}(t_i), t_i) \tag{10.12b}$$

where k is the (integer) ratio of the two step sizes, $j = 0 \ldots k-1$, and $h = t_{i+1} - t_i$ is the step–size of the slow sub–system.

Equations (10.12a–b) do not specify, how $\mathbf{x}_\mathbf{s}(t_i + j\cdot h)$ is being calculated, since the variables of the slow sub–system are not evaluated at the intermediate time instants.

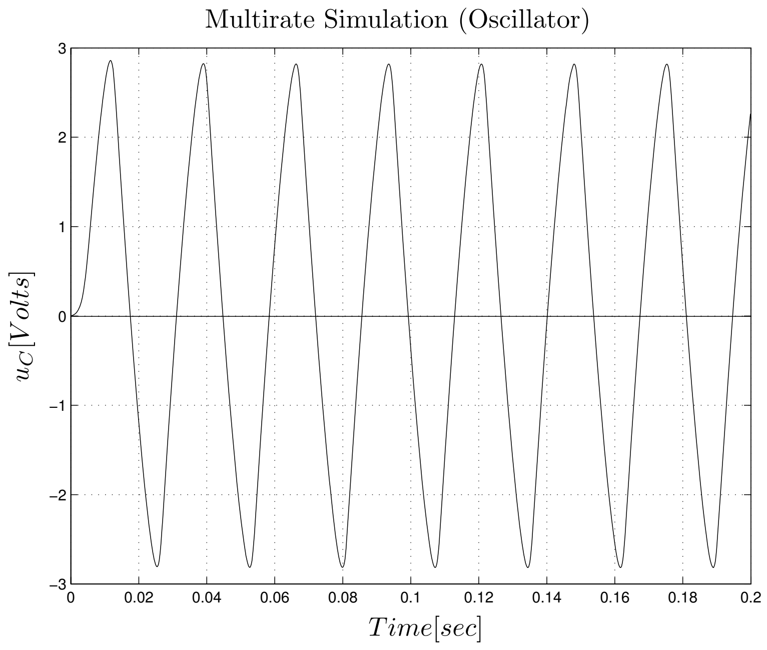

FIGURE 10.3. Van–der–Pol oscillator voltage.

In our example, we chose $\mathbf{x_s}(t_i + j \cdot h) = \mathbf{x_s}(t_i)$, i.e., we used the last calculated value. A more accurate solution might involve using some form of extrapolation technique.

This last problem is known as the *interfacing* problem [10.30]. It is related to the way, in which the fast and slow sub–systems are interconnected with each other.

In our case, we used the Forward Euler method. Similar approaches have been reported in the literature based on the 2^{nd}–order explicit Adams–Bashforth technique [10.24], including also some improvements for parallel implementation.

In spite of the improvement achieved in this case using multi–rate integration, we must not forget that the example we analyzed was not very demanding, since the speed of the fast sub–system is not much higher than that of the slow sub–system. We already know that explicit algorithms won't work in more strongly stiff systems.

In those cases, as previously discussed, semi–implicit methods may be a better choice in the real–time context. However, we know that in large systems, those methods have a drawback, as they need to invert a potentially very large matrix.

A solution that combines both ideas, multi–rate and semi–implicit integration, consists in splitting the system into a fast and a slow part, while applying a semi–implicit method to the fast sub–system, whereas the

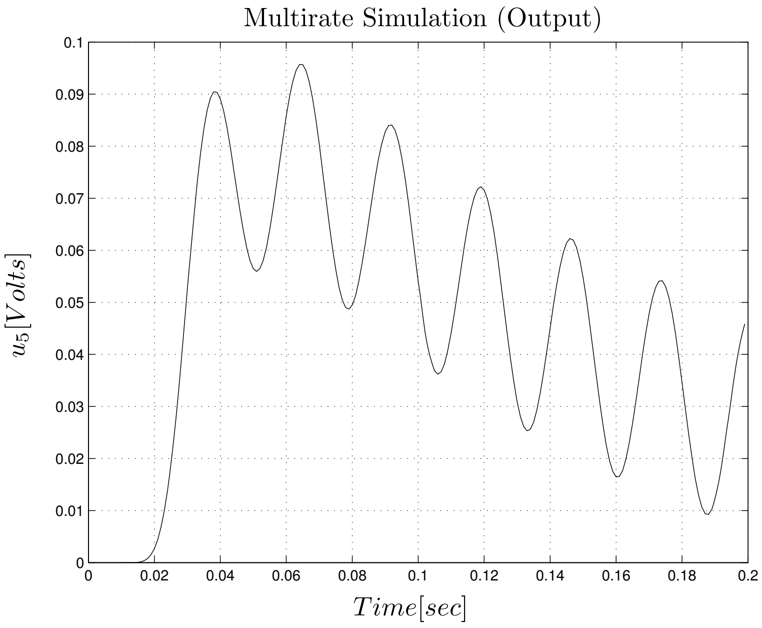

FIGURE 10.4. Transmission line output voltage.

slow sub–system is being simulated using an explicit integration algorithm. These types of schemes are referred to in the literature as *mixed–mode* integration algorithms.

We shall discuss mixed–mode integration in due course, but let us first pursue another avenue.

10.6 Inline Integration

Figure 10.5 shows the same circuit as Fig. 10.2 with the inclusion of an additional RC load at the end of the transmission line.

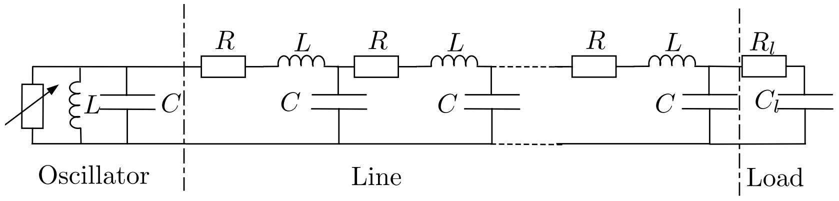

FIGURE 10.5. Van–der–Pol oscillator and transmission line.

The state equations are similar to the previous case, but now we have:

$$\begin{aligned}
\frac{di_L}{dt} &= \frac{1}{L}u_C && \text{(10.13a)}\\
\frac{du_C}{dt} &= \frac{1}{C}(u_C - k \cdot u_C^3 - i_L - i_1) && \text{(10.13b)}\\
\frac{di_1}{dt} &= \frac{1}{L}u_C - \frac{R}{L}i_1 - \frac{1}{L}u_1 && \text{(10.13c)}\\
\frac{du_1}{dt} &= \frac{1}{C}i_1 - \frac{1}{C}i_2 && \text{(10.13d)}\\
\frac{di_2}{dt} &= \frac{1}{L}u_1 - \frac{R}{L}i_2 - \frac{1}{L}u_2 && \text{(10.13e)}\\
\frac{du_2}{dt} &= \frac{1}{C}i_2 - \frac{1}{C}i_3 && \text{(10.13f)}\\
&\vdots\\
\frac{di_n}{dt} &= \frac{1}{L}u_{n-1} - \frac{R}{L}i_n - \frac{1}{L}u_n && \text{(10.13g)}\\
\frac{du_n}{dt} &= \frac{1}{C}i_n - \frac{1}{R_l \cdot C}(u_n - u_l) && \text{(10.13h)}\\
\frac{du_l}{dt} &= \frac{1}{R_l \cdot C_l}(u_n - u_l) && \text{(10.13i)}
\end{aligned}$$

Let us assume that the load parameters are $R_l = 1\ \text{k}\Omega$ and $C_l = 1\ \text{nF}$.

Since the load resistor is much bigger than the line resistors, the newly introduced term in Eq.(10.13h) won't influence the dynamics of the transmission line significantly, and we can expect the sub–system (10.13a–h) to exhibit a similar behavior to the one of System (10.10).

However, the last state equation, Eq.(10.13i), introduces a fast pole. The position of this pole is approximately located at:

$$\lambda_l \approx -\frac{1}{R_l \cdot C_l} = -10^6\ \text{sec}^{-1} \tag{10.14}$$

on the negative real axis of the complex λ–plane.

This means that we would have to reduce the step size by about a factor of 1000 with respect to the previous example, in order to obtain a numerically stable result.

Unfortunately, such a solution is completely unacceptable in the context of a real–time simulation.

A first alternative might be to replace the Forward Euler algorithm by the semi–implicit Euler method studied earlier in this chapter. However, this is a system of order 13, and, leaving superstitions aside, we may not have the luxury of inverting a 13×13 matrix at each step.

A second alternative might be to inline the Backward Euler algorithm [10.16] and apply the tearing method to the resulting set of difference equations. Let us rewrite the model using the inling approach.

$$\begin{aligned}
i_L &= \text{pre}(i_L) + \frac{h}{L}u_C && \text{(10.15a)}\\
u_C &= \text{pre}(u_C) + \frac{h}{C}(u_C - k \cdot u_C^3 - i_L - i_1) && \text{(10.15b)}\\
i_1 &= \text{pre}(i_1) + \frac{h}{L}u_C - \frac{Rh}{L}i_1 - \frac{h}{L}u_1 && \text{(10.15c)}\\
u_1 &= \text{pre}(u_1) + \frac{h}{C}i_1 - \frac{h}{C}i_2 && \text{(10.15d)}\\
i_2 &= \text{pre}(i_2) + \frac{h}{L}u_1 - \frac{Rh}{L}i_2 - \frac{h}{L}u_2 && \text{(10.15e)}\\
u_2 &= \text{pre}(u_2) + \frac{h}{C}i_2 - \frac{h}{C}i_3 && \text{(10.15f)}\\
&\vdots\\
i_n &= \text{pre}(i_n) + \frac{h}{L}u_{n-1} - \frac{Rh}{L}i_n - \frac{h}{L}u_n && \text{(10.15g)}\\
u_n &= \text{pre}(u_n) + \frac{h}{C}i_n - \frac{h}{R_l \cdot C}(u_n - u_l) && \text{(10.15h)}\\
u_l &= \text{pre}(u_l) + \frac{h}{R_l \cdot C_l}(u_n - u_l) && \text{(10.15i)}
\end{aligned}$$

The causalized structure digraph is shown in Fig. 10.6.

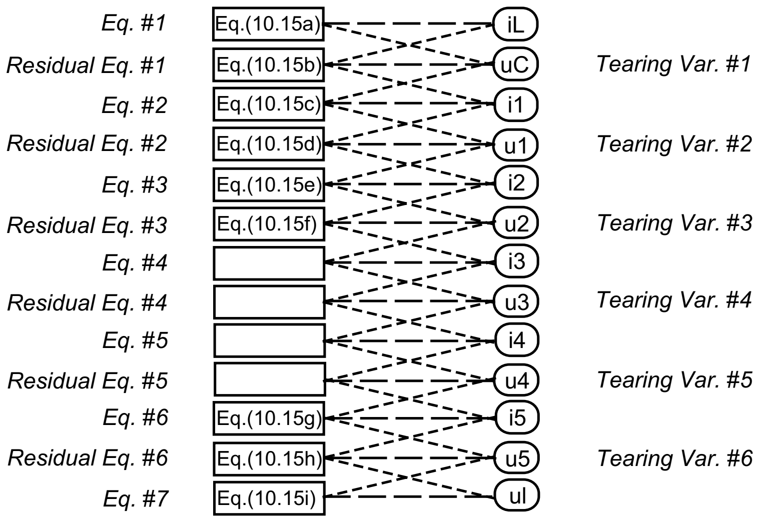

FIGURE 10.6. Causal structure diagram of electrical circuit.

Inlining did help indeed. We got away with six tearing variables. Instead of having to invert a 13×13 matrix in every step, we now must invert a

6×6 matrix. Since even the best linear sparse matrix solver grows at least quadratically with the size of the system in terms of its computational complexity, the savings were quite dramatic. The computations just got faster by about a factor of four.

Although inline integration had been developed for general simulation problems, it turns out that this method has become a quite powerful ally in dealing with real–time simulation as well [10.15].

But what, if the simulation is still too slow? What if the transmission line consists of 50 segments, instead of only 5 of them? Mixed–mode integration may be the answer to our needs.

10.7 Mixed–mode Integration

A more careful look at the system shows that there is no strong interaction between the subsystems of Eqs.(10.13a–h) and Eq.(10.13i). In fact, the fast dynamics can be explained by looking at the last equation alone.

Thus, it might be reasonable to use Backward Euler (or semi–implicit Euler) only in the last equation.

To this end, we inlined the equations once more, this time using the explicit Forward Euler algorithm everywhere except for the last equation, where we still used the implicit Backward Euler method.

The resulting inlined difference equation system no can be written as follows:

$$\begin{aligned}
i_L &= \mathrm{pre}(i_L) + \frac{h}{L}\mathrm{pre}(u_C) && (10.16\text{a})\\
u_C &= \mathrm{pre}(u_C) + \frac{h}{C} = [\mathrm{pre}(u_C) - k \cdot \mathrm{pre}(u_C)^3 - \mathrm{pre}(i_L) \\
&\quad -\mathrm{pre}(i_1)] && (10.16\text{b})\\
i_1 &= \mathrm{pre}(i_1) + \frac{h}{L}\mathrm{pre}(u_C) - \frac{Rh}{L}\mathrm{pre}(i_1) - \frac{h}{L}\mathrm{pre}(u_1) && (10.16\text{c})\\
u_1 &= \mathrm{pre}(u_1) + \frac{h}{C}\mathrm{pre}(i_1) - \frac{h}{C}\mathrm{pre}(i_2) && (10.16\text{d})\\
i_2 &= \mathrm{pre}(i_2) + \frac{h}{L}\mathrm{pre}(u_1) - \frac{Rh}{L}\mathrm{pre}(i_2) - \frac{h}{L}\mathrm{pre}(u_2) && (10.16\text{e})\\
u_2 &= \mathrm{pre}(u_2) + \frac{h}{C}\mathrm{pre}(i_2) - \frac{h}{C}\mathrm{pre}(i_3) && (10.16\text{f})\\
&\vdots \\
i_n &= \mathrm{pre}(i_n) + \frac{h}{L}\mathrm{pre}(u_{n-1}) - \frac{Rh}{L}\mathrm{pre}(i_n) - \frac{h}{L}\mathrm{pre}(u_n) && (10.16\text{g})\\
u_n &= \mathrm{pre}(u_n) + \frac{h}{C}\mathrm{pre}(i_n) - \frac{h}{R_l \cdot C}\left[\mathrm{pre}(u_n) - \mathrm{pre}(u_l)\right] && (10.16\text{h})
\end{aligned}$$

$$u_l = \text{pre}(u_l) + \frac{h}{R_l \cdot C_l}(u_n - u_l) \tag{10.16i}$$

All equations are now explicit, except for the very last equation, Eq.(10.16i), which is implicit in the variable u_l. Furthermore, Eq.(10.16i) can only be computed after $u_n(t)$ has been evaluated first from Eq.(10.16h). Thus, the size of the Jacobian is now 1×1.

We simulated the system using the same approach as before, i.e., we applied a step size of 10^{-4} seconds to the two oscillator equation, whereas we used a step size of 10^{-3} seconds on all of the other equations, including the implicit load equation. The simulation results are shown in Figure 10.7.

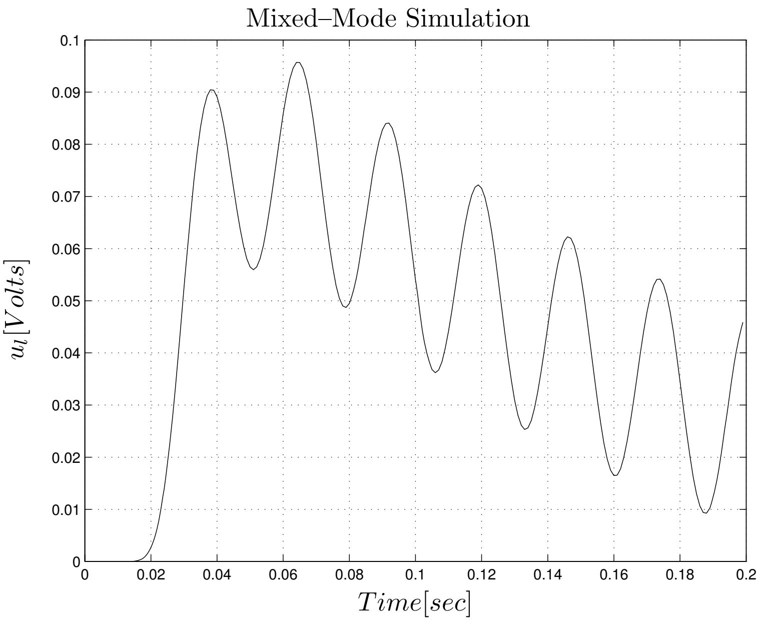

FIGURE 10.7. Load output voltage.

In more general terms, given a system like Eqs.(10.11a–b), the Backward–Forward Euler Mixed–Mode integration scheme is given by the formula:

$$\mathbf{x_s}(t_{k+1}) = \mathbf{x_s}(t_k) + h \cdot \mathbf{f_s}(\mathbf{x_f}(t_k), \mathbf{x_s}(t_k), t_k) \tag{10.17a}$$

$$\mathbf{x_f}(t_{k+1}) = \mathbf{x_f}(t_k) + h \cdot \mathbf{f_f}(\mathbf{x_f}(t_{k+1}), \mathbf{x_s}(t_{k+1}), t_{k+1}) \tag{10.17b}$$

Thus, the algorithm starts by computing explicitly the value of $\mathbf{x_s}(t_{k+1})$. It then uses this value to evaluate $\mathbf{x_f}(t_{k+1})$ either implicitly or in a semi–implicit fashion.

Mixed–mode integration as presented in this section was first introduced by Krebs [10.29] for an entirely different purpose, namely to resolve the problem of *conditional index changes* once and for all in a systematic and algorithmic fashion.

The technique was rediscovered independently by Schiela [10.39] for the purpose of speeding up real–time simulation. Schiela proposed the use of linearization and eigenvalue analysis to discern, which of the integrators should be inlined using Forward Euler, and which should be inlined using Backward Euler, i.e., for determining the slow and fast sub–systems.

The advantage of solving the implicit equation only for the components $\mathbf{x_f}(t_{k+1})$ can turn out to be very important in systems, such as the one presented here, where the length of vector $\mathbf{x_f}$ is considerably smaller than that of $\mathbf{x_s}$.

In our (rather academic) example, the reduction in the number of calculations is huge. In more realistic applications, the literature reports speed–up factors of 4 to 16 [10.39].

Mixed–mode versions of higher order Runge–Kutta methods and approaches also combine mixed–mode and multi–rate integration techniques have also been reported in the literature [10.42].

In fact, we used a mixture of multi–rate and mixed–mode integration in our example, as we used a ten times smaller step size of 10^{-4} seconds for the integration of the two oscillator equations.

Both multi–rate integration and mixed–mode integration assume that there indeed exist two distinct and discernable sub–systems. This may not always be the case. For example, the real–time simulation of a distributed parameter system described by a parabolic PDE, such as for the purpose of optimal control of a space heating system, does not share this property. The eigenvalues are simply spread out. Also, if a system is highly nonlinear, the concept of looking at eigenvalues by itself become dubious, as eigenvalues can only be defined for the linearized system. In a sufficiently nonlinear system, the eigenvalues of the linearized system move around as a function of time, which again may prevent us from subdividing the system into two distinct and time–invariant sub–systems, one fast, the other slow.

10.8 Discontinuous Systems

Real–time simulation of discontinuous models is highly problematic, as state events happen asynchronously. Event handling invariably causes overhead that needs to be accounted for. Thus, if until now, it may have been acceptable to have the computations associated with the simulation of a single step occupy somewhere around 80% of the allotted time, we can no longer do so if the model to be simulated is discontinuous. In the case of discontinuous models being simulated in real time, it is prudent to dimension the computer system such that regular steps occupy no more than about 20% of the allotted time. This will grant us the additional time needed to handle no more than one state event per step.

State–event handling in real–time simulation is simplified, when comparing it to the techniques introduced in the previous chapter, by two factors:

1. As we are using low–order integration techniques, we can also use low–order event localization algorithms.

2. Since we use much smaller step sizes, the precise localization of state events becomes less critical, and there shouldn't occur as often multiple state events within a single integration step.

Since we must control the total amount of computations performed within an integration step, iterative techniques for localizing state events are out. We must rely on interpolation alone.

Yet, as we are using low–order integration techniques, the former iterative algorithms can now be employed as interpolators. For example, if we integrate by inlining a first–order accurate algorithm, i.e., either Forward Euler or Backward Euler [10.16], we can use a single step of *Regula Falsi* to locate the event as accurately as we can hope to accomplish with such a crude integration algorithm. If we decide to inline the third–order accurate Radau–IIA algorithm [10.5, 10.7], a single step of cubic interpolation will localize the discontinuity as accurately as can be done using such an integration method.

Of course, it may be possible to reduce the residual on the zero–crossing function further by iteration, but this does not necessarily imply that we would thereby locate the event more accurately, as already the previous integration steps are contaminated by numerical errors.

Let us discuss, how event handling may proceed. We start out by performing a regular integration step, advancing the simulation from time t_n to time t_{n+1}. At the end of the step, we discover that a zero crossing has taken place. We interpolate to the next event time, t_{next}. Since we don't have dense output [10.13] available, as this would be too expensive to compute in real time, we shall have to repeat the last integration step to advance the entire state vector from time t_n to time t_{next}. We then perform the actions associated with the event, and compute a new consistent initial state. Starting from that new initial state, we perform another partial state advancing the state vector from time t_{next} to time t_{n+1}. The solution obtained in this way can then be pushed out through the D/A–converters and communicated back to whatever hardware needs it.

As no iteration takes place, the amount of work, i.e., the total number of floating–point operations needed, can be estimated accurately. Assuming that only one state event is allowed to occur within a single integration step, we can thus calculate, how much extra time we need to allot, in order to handle single state events within an integration step adequately.

Unfortunately, the extra amount of work for event handling is non–negligible. We perform three integration steps instead of only one, and we have to accommodate the additional computations needed to process the event actions themselves. Thus, the total effort grows by about a factor of four. This is the reason, why we wrote earlier that the allowed resource utilization for regular integration steps needs to drop from about 80% to about 20%.

10.9 Simulation Architecture

We haven't yet discussed, how the simulation engine is physically connected to the hardware. Although it would be possible to connect directly the output signals of the *sensor units* with the input of the *A/D–converters*, which form part of the simulation engine, and the outputs of the *D/A–converters*, also integrated with the simulation engine, with the input signals of the *actuator units*, this is hardly ever done in today's world.

Instead, commercial converters have their own computer chips built in, that perform the necessary computations and store the digital signals in *mailboxes*. Thus, an A/D–converter is really a converter together with a built–in *zero–order hold (ZOH)* unit. Once the analog signal has been converted, it is available for whichever process needs it, until it is overridden by the next *sample–and–hold (S/H)* cycle. A D/A–converter doesn't take its data from the simulation directly, but instead, takes it out of its own mailbox. Even the sensor and actuator units contain their own hardware–built sample–and–hold equipment.

Handshaking mechanisms are needed to prevent the simulator from replacing the data in the mailbox of the D/A–converter, while the converter tries to read out the data from its own mailbox. Similarly, handshaking mechanisms are needed to prevent the simulator from reading the data from the mailbox of the A/D–converter, while these data are in the process of being updated by the converter.

A possible physical configuration of a *hardware–in–the–loop (HIL)* simulation is shown in Fig. 10.8.

Protocols have been designed to ensure that these handshaking mechanisms always work correctly. To this end, the *High–Level Architecture (HLA)* standard was created in the U.S. [10.44], whereas Europe developed its own standard with CORBA [10.37].

Consequently, Fig. 10.1 needs to be modified. The time needed for the A/D–convertions and D/A–conversions are no longer part of the computational load associated with advancing the simulation by one time step, as these activities are performed in parallel by separate units. Instead, we must include the time needed for the read and write requests from and to the mailboxes across the architecture.

Since the total time needed for computing all activities associated with a single integration step must be known, both HLA and CORBA offer

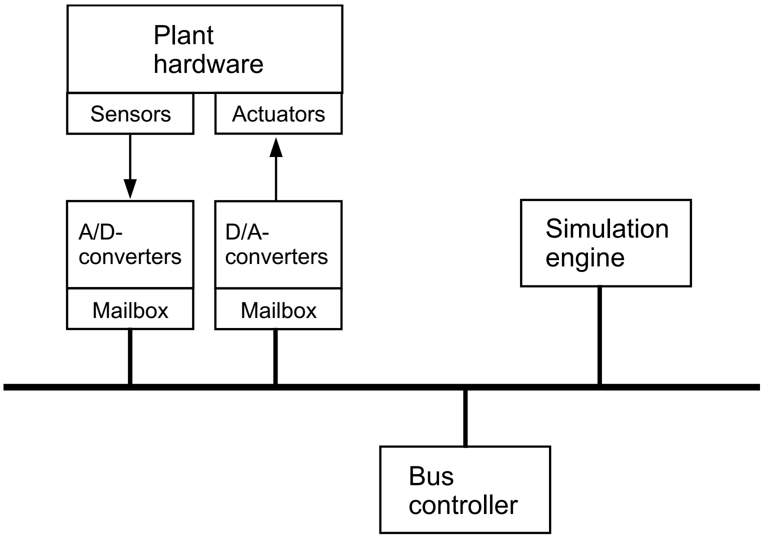

FIGURE 10.8. Physical configuration of HIL simulation.

mechanisms for specifying the *maximum allowed latency* in answering requests for information transfer across the architecture using the established communication channels and protocols.

10.10 Overruns

Overruns are defined as situations, where, in spite of our best efforts, the simulation engine is unable to perform all of the required computations in time to advance its state to the next clock time, before the real–time clock interrupt is received.

This may happen, because it cannot be guaranteed that no more than one state event will ever occur within a single integration step. As all events must be processed, it can happen that the simulation falls behind. Most real–time simulations specify the maximum percentage of overruns as e.g. 1% or 2%.

What happens, when the simulation falls behind? Thanks to the buffers implemented in the form of the mailboxes, the hardware will hardly notice it. It simply receives the same actuator values for a second time in a row.

For the simulation software, the situation may be worse, because it may need to know, what time it is. Thus, the following procedure is recommended in the case of an overrun. If the next real–time interrupt arrives, before the computations have been completed, the subsequent integration step is doubled in length to catch up with real time. In this way, we allow one integration step to be computed less accurately once in a while, in order to stay synchronous with the real–time clock.

10.11 Summary

In this chapter, we have attempted to paint, using a fairly wide brush, a picture of some of the requirements associated with real–time simulation of physical systems. It's a difficult problem to cope with, as the information available on this topic is widely scattered in the literature and hardly available in a concise and consistent fashion.

Just like in the case of the distributed parameter systems, we do not claim that we have been able to create here a body of knowledge that is exhaustive by any standard. We do not claim that you, the reader, will be able to successfully build a real–time simulator after having read this chapter.

Naturally, as this book concerns itself primarily with topics surrounding the numerical integration of ODE and DAE systems, we have focused our emphasis on issues related to the special demands of real–time simulation on the integration algorithms.

We only just mentioned the available literature on simulation speed–up by means of efficient function generation [10.21, 10.28], and we didn't talk at all about the use of special–purpose simulation hardware, a fashionable topic in the 1960's to 1980's. These systems have largely been overcome by events, as conventional digital hardware became faster and faster.

We barely scratched the surface of issues concerning the *simulation architecture.* There is a substantial body of knowledge available on this subject, although it concerns itself more with *discrete event simulation* in general, than with *physical system simulation.*

We didn't even mentioned the topic of *distributed real–time simulation* [10.8, 10.31], where the execution speed of the real–time simulation is increased by distributing the computations necessary to complete an integration step over multiple computers communicating with each other across the simulation architecture.

Notice that even Fig. 10.8 does not reflect the full real–time architecture needed to perform distributed simulation experiments. Both HLA and CORBA were developed to support *distributed processing.* Whereas CORBA was designed primarily for *instrumentation*, HLA has a strong emphasis on *distributed simulation.* In Fig. 10.8, the simulation is still "in charge" of the overall operations. All other units are essentially subservient to the simulation.

If we allow the simulation to be distributed over multiple processors working in parallel on a demanding simulation task, this approach won't work any longer. Figure 10.9 shows a once more enhanced architecture that supports distributed simulation.

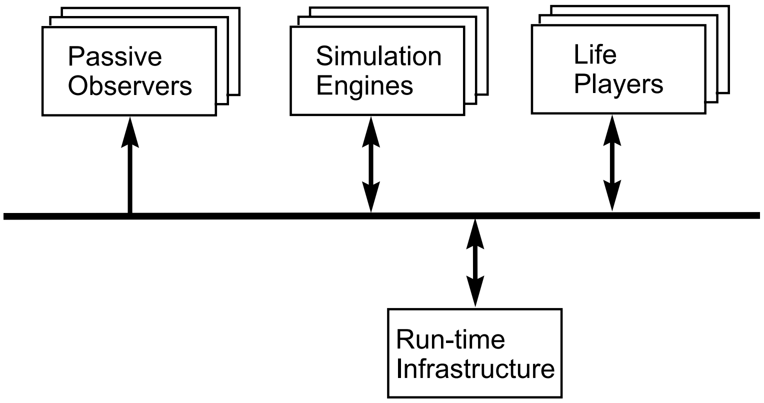

FIGURE 10.9. The HLA architecture.

Figure 10.9 depicts the overall HLA architecture [10.9, 10.10] for distributed simulation. Here, the former *bus controller* is replaced by the *Real-time Infrastructure (RTI)* [10.20], a distributed operating system that coordinates the activities of the various participants in the simulation. Each participant is responsible for finishing its assigned tasks within the allotted time slot and for returning the results in a timely fashion to the RTI.

¿From the perspective of the architecture, there is essentially no difference between *simulators* and *life players*, i.e., hardware–in–the–loop. *Passive observers* were added as an additional type of participants. Since passive observers never return any data to the architecture, it is not essential that they operate in a time–synchronous fashion. They can complete their tasks on an "as–fast–as–possible" basis.

10.12 References

[10.1] Jeff R. Cash. A Semi–implicit Runge–Kutta Formula for the Integration of Stiff Systems of Ordinary Differential Equations. *Chemical Engineering J.*, 20(3):219–224, 1980.

[10.2] François E. Cellier, Larry C. Schooley, Malur K. Sundareshan, and Bernard P. Zeigler. Computer–aided Design of Intelligent Controllers: Challenge of the Nineties. In *Recent Advances in Computer Aided Control Systems Engineering*, pages 53–80, Amsterdam, the Netherlands, 1992. Elsevier Science Publishers.

[10.3] François E. Cellier, Larry C. Schooley, Bernard P. Zeigler, Adele Doser, Glenn Farrenkopf, JinWoo Kim, YaDung Pan, and Brian Willams. Watchdog Monitor Prevents Martian Oxygen Production Plant from Shutting Itself Down During Storm. In *Proceedings IS-*

RAM'92, ASME Conference on Intelligent Systems for Robotics and Manufacturing, pages 697–704, Santa Fe, N.M., 1992.

[10.4] François E. Cellier. *Continuous System Modeling.* Springer Verlag, New York, 1991. 755p.

[10.5] François E. Cellier. Inlining Step–size Controlled Fully Implicit Runge–Kutta Algorithms for the Semi–analytical and Semi–numerical Solution of Stiff ODEs and DAEs. In *Proceedings* 5th *Conference on Computer Simulation*, pages 259–262, Mexico City, Mexico, 2000.

[10.6] Richard J. Charron and Min Hu. A–contractivity of Linearly Implicit Multistep Methods. *SIAM Journal on Numerical Analysis*, 32(1):285–295, 1995.

[10.7] Christoph Clauss, Hilding Elmqvist, Sven Erik Mattsson, Martin Otter, and Peter Schwarz. Mixed Domain Modeling in Modelica. In *Proceedings FDL'02, Forum on Specification and Design Languages*, Marseille, France, 2002.

[10.8] Rémi Cozot. From Multibody Systems Modeling to Distributed Real–Time Simulation. In *Proceedings Simulation'96 IEEE Conference*, pages 234–241, 1996.

[10.9] Judith S. Dahmann, Frederick Kuhl, and Richard Weatherly. Standards for Simulation: As Simple As Possible But Not Simpler – The High Level Architecture For Simulation. *Simulation*, 71(6):378–387, 1998.

[10.10] Judith S. Dahmann. The High Level Architecture and Beyond: Technology Challenges. In *Proceedings PADS'99,* 13th *Workshop on Parallel and Distributed Simulation*, pages 64–70, Atlanta, Georgia, 1999.

[10.11] Álvaro de Albornoz Bueno and François E. Cellier. Qualitative Simulation Applied to Reason Inductively About the Behavior of a Quantitatively Simulated Aircraft Model. In *Proceedings QUARDET'93, IMACS International Workshop on Qualitative Reasoning and Decision Technologies*, pages 711–721, Barcelona, Spain, 1993.

[10.12] Álvaro de Albornoz Bueno and François E. Cellier. Variable Selection and Sensor Fusion in Automatic Hierarchical Fault Monitoring of Large Scale Systems. In *Proceedings QUARDET'93, IMACS International Workshop on Qualitative Reasoning and Decision Technologies*, pages 722–734, Barcelona, Spain, 1993.

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

[10.14] Eduard Eitelberg. *Modellreduktion linearer zeitinvarianter Systeme durch Minimieren des Gleichungsfehlers.* PhD thesis, University of Karlsruhe, Karlsruhe, Germany, 1979.

[10.15] Hilding Elmqvist, Sven Erik Mattsson, and Hans Olsson. New Methods for Hardware–in–the–loop Simulation of Stiff Models. In *Proceedings Modelica'2002 Conference*, pages 59–64, Oberpfaffenhofen, Germany, 2002.

[10.16] Hilding Elmqvist, Martin Otter, and François E. Cellier. Inline Integration: A New Mixed Symbolic/Numeric Approach for Solving Differential–Algebraic Equation Systems. In *Proceedings European Simulation Multiconference*, pages xxiii–xxxiv, Prague, Czech Republic, 1995.

[10.17] Hilding Elmqvist and Martin Otter. Methods for Tearing Systems of Equations in Object–oriented Modeling. In *Proceedings European Simulation Multiconference*, pages 326–332, Barcelona, Spain, 1994.

[10.18] Javier Garcia de Jalón and Eduardo Bayo. *Kinematic and Dynamic Simulation of Multibody Systems –The Real–Time Challenge–.* Wiley, 1994.

[10.19] Kjell Gustafsson and Gustaf Söderlind. Control Strategies for the Iterative Solution of Nonlinear Equations in ODE Solvers. *SIAM Journal on Scientific Computing*, 18(1):23–40, 1997.

[10.20] Frank Hodum and David Edwards. Time Management Services in the RTI–NG. In *Proceedings SIW'01, Fall Simulation Interoperability Workshop*, paper 01F–SIW–090, 2001.

[10.21] Robert M. Howe and Kuo-Chin Lin. The Use of Function Generation in the Real–time Simulation of Stiff Systems. In *AIAA Flight Simulation Technologies Conference and Exhibit*, pages 217–224, Dayton, Ohio, 1990.

[10.22] Robert M. Howe. The Use of Mixed Integration Algorithms in State Space. *Transactions of the Society for Computer Simulation*, 7(1):45–66, 1990.

[10.23] Robert M. Howe. On–line Calculation of Dynamic Errors in Real–time Simulation. In *Proceedings of SPIE*, volume 3369, pages 31–42, 1998.

[10.24] Robert M. Howe. Real–Time Multi–Rate Asynchronous Simulation with Single and Multiple Processors. In *Proceedings of SPIE*, volume 3369, pages 3–14, 1998.

[10.25] Thomas Kailath. *Linear Systems.* Prentice–Hall, Englewood Cliffs, N.J., 1980.

[10.26] Granino A. Korn and John V. Wait. *Digital Continuous–System Simulation.* Prentice–Hall, Englewood Cliffs, N.J., 1978.

[10.27] Granino A. Korn. *Interactive Dynamic–System Simulation.* McGraw–Hill, New York, 1989.

[10.28] Granino A. Korn. *Neural–Network Experiments on Personal Computers.* MIT Press, Cambridge, Mass., 1991.

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

[10.30] John Laffitte and Robert M. Howe. Interfacing Fast and Slow Subsystems in the Real–time Simulation of Dynamic Systems. *Transactions of SCS*, 14(3):115–126, 1997.

[10.31] Nicolas Léchevin, Camille Alain Rabbath, and Paul Baracos. Distributed Real–time Simulation of Power Systems Using Off–the–shelf Software. *IEEE Canadian Review*, pages 5–8, 2001. summer edition.

[10.32] Kuo-Chin Lin and Robert M. Howe. Simulation Using Staggered Integration Steps — Part I: Intermediate–Step Predictor Methods. *Transactions of the Society for Computer Simulation*, 10(3):153–164, 1993.

[10.33] Kuo-Chin Lin and Robert M. Howe. Simulation Using Staggered Integration Steps — Part II: Implementation on Dual–Speed Systems. *Transactions of the Society for Computer Simulation*, 10(4):285–297, 1993.

[10.34] Francisco Mugica and François E. Cellier. Automated Synthesis of a Fuzzy Controller for Cargo Ship Steering by Means of Qualitative Simulation. In *Proceedings ESM'94, European Simulation MultiConference*, Barcelona, Spain, 1994.

[10.35] Martin Otter, Hilding Elmqvist, and François E. Cellier. 'Relaxing' – A Symbolic Sparse Matrix Method Exploiting the Model Structure in Generating Efficient Simulation Code. In *Proceedings Symposium on Modeling, Analysis, and Simulation, CESA'96, IMACS Multi-Conference on Computational Engineering in Systems Applications*, volume 1, pages 1–12, Lille, France, 1996.

[10.36] Olgierd A. Palusinski. Simulation of Dynamic Systems Using Multirate Integration Techniques. *Transactions of SCS*, 2(4):257–273, 1986.

[10.37] José Ignacio Rodríguez, José Manuel Jiménez, Francisco Javier Funes, and Javier Garcia de Jalón. Dynamic Simulation of Multi-Body Systems on Internet Using CORBA, Java and XML. *Multibody System Dynamics*, 10(2):177–199, 2003.

[10.38] Anton Schiela and Folkmar Bornemann. Sparsing in Real Time Simulation. *ZAMM, Zeitschrift für angewandte Mathematik und Mechanik*, 83(10):637–647, 2003.

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

[10.40] Anton Schiela. Sparsing in Real Time Simulation. Diploma Project, Technische Universität München, 2002. 75p.

[10.41] Michael C. Schweisguth and François E. Cellier. A Bond Graph Model of the Bipolar Junction Transistor. In *Proceedings SCS* 4th *International Conference on Bond Graph Modeling and Simulation*, pages 344–349, San Francisco, California, 1999.

[10.42] Siddhartha Shome. *Dual–rate Integration Using Partitioned Runge–Kutta Methods for Mechanical Systems With Interacting Subsystems.* PhD thesis, The University of Iowa, 2000.

[10.43] Jon M. Smith. *Mathematical Modeling and Digital Simulation for Engineers and Scientists.* John Wiley & Sons, New York, second edition, 1987.

[10.44] Simon J.E. Taylor, Jon Saville, and Rajeev Sudra. Developing Interest Management Techniques in Distributed Interactive Simulation Using Java. In *Proceedings WSC'99, Winter Simulation Conference*, pages 518–523, 1999.

[10.45] Pentti J. Vesanterä and François E. Cellier. Building Intelligence into an Autopilot – Using Qualitative Simulation to Support Global Decision Making. *Simulation*, 52(3):111–121, 1989.

[10.46] Jörg Wensch, Karl Strehmel, and Rüdiger Weiner. A Class of Linearly–Implicit Runge–Kutta Methods for Multibody Systems. *Applied Numerical Mathematics*, 22(1–3):381–398, 1996.

[10.47] Bernard P. Zeigler, Larry C. Schooley, François E. Cellier, and FeiYue Wang. High–Autonomy Control of Space Resource Processing Plants. *IEEE Control Systems*, 13(3):29–39, 1993.

10.13 Bibliography

[B10.1] Hilding Elmqvist, Sven Erik Mattsson, Hans Olsson, Johan Andreasson, Martin Otter, Christian Schweiger, and Brück, Dag . Real–time Simulation of Detailed Automotive Models. In *Proceedings 2003 Modelica Conference*, Linköping, Sweden, 2003.

[B10.2] José Manuel Jiménez Bascones. *Formulaciones cinemáticas y dinámicas para la simulación en tiempo real de sistemas de sólidos rígidos.* PhD thesis, Universidad de Navarra, San Sebastián, Spain, 1993.

[B10.3] Sean Murphy, Jonathan Labin, and Robert Lutz. Experiences Using the Six Services of the IEEE 1516.1 Specification: A 1516 Tutorial. In *Proceedings SIW'04, Spring Simulation Interoperability Workshop*, paper 04S–SIW–056, 2004.

[B10.4] Shinichi Soejima and Takashi Matsuba. Application of Mixed Mode Integration and New Implicit Inline Integration at Toyota. In *Proceedings 2002 Modelica Conference*, Oberpfaffenhofen, Germany, 2002.

10.14 Homework Problems

[H10.1] Semi–implicit Trapezoidal Rule

Derive a semi–implicit version of the trapezoidal rule

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_k) + \frac{h}{2}[\mathbf{f}(\mathbf{x}(t_k), t_k) + \mathbf{f}(\mathbf{x}(t_{k+1}), t_{k+1})] \qquad \text{(H10.1a)}$$

Hint: Approximate $\mathbf{f}(\mathbf{x}(t_{k+1}), t_{k+1})$ using the ideas developed in Section 10.4.

Show that the stability domain of the method coincides with that of the fully–implicit trapezoidal rule (i.e., show that also the semi–implicit trapezoidal rule is F–stable).

[H10.2] Pendulum

Using the semi–implicit trapezoidal formula, simulate a pendulum motion that can be described by the state–space model:

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -\sin(x_1) - b \cdot x_2 \end{aligned}$$

assuming that the friction parameter is $b = 0.02$.

Start from the initial condition $x_0 = (0.5\ 0.5)^T$ using a step size of $h = 0.5$, and simulate until $t_f = 500$ time units.

Repeat the experiment using the fully–implicit trapezoidal rule.

Compare the results as well as the number of floating–point operations required for the two simulations.

[H10.3] Frictionless Pendulum

Repeat problem H10.2 with $b = 0$ (i.e., without friction).

Compare the results obtained with the two integration methods. Explain the differences.

[H10.4] Hydraulic Motor

We wish to simulate a position control system involving a hydraulic motor. Figure H10.4a shows the schematic of a hydraulic motor with two chambers and a set of flows.

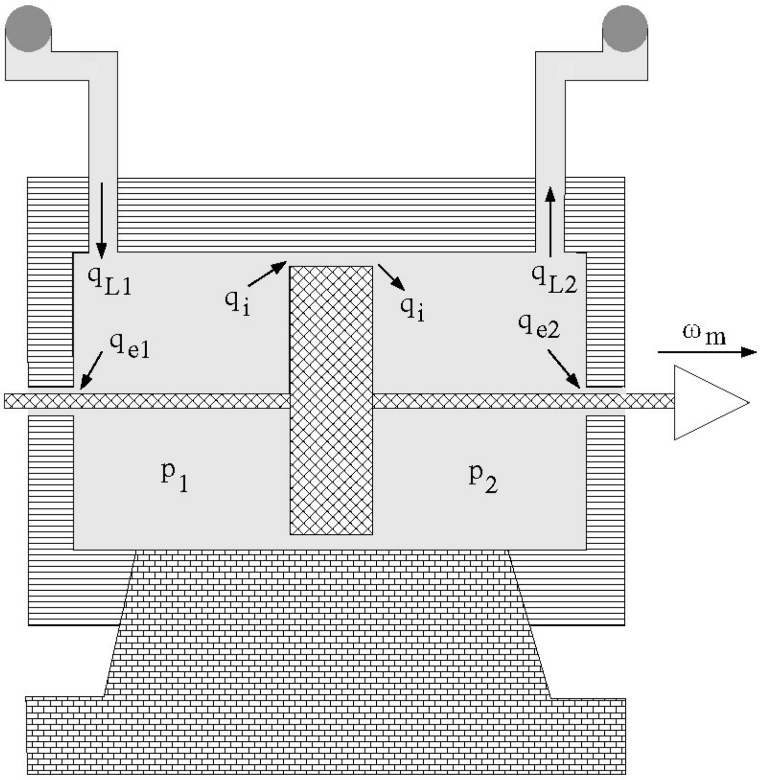

FIGURE H10.4a. Hydraulic motor schematic.

The physics behind the hydraulic motor model are explained in the companion book [10.4].

Due to the compressibility of the fluid, the change in the pressures of the two chambers is proportional to the flow balance in and out of these chambers:

$$\dot{p}_1 = c_1 \cdot (q_{L1} - q_i - q_{e1} - q_{ind}) \tag{H10.4a}$$

$$\dot{p}_2 = c_1 \cdot (q_{ind} + q_i - q_{e2} - q_{L2}) \tag{H10.4b}$$

where $c_1 = 5.857 \times 10^{13}$ kg m^{-4} sec^{-2} is the inverse of the compressibility constant.

There exist several laminar leakage flows in this model. The flow q_i is the internal leakage flow between the two chambers:

$$q_i = c_i \cdot p_L = c_i \cdot (p_1 - p_2) \tag{H10.4c}$$

where p_L is the load pressure of the motor, and $c_i = 0.737 \times 10^{-13}$ kg^{-1} m^4 sec is the internal leakage coefficient. The flows q_{e1} and q_{e2} are external leakage flows:

$$q_{e1} = c_e \cdot (p_1 - p_0) \tag{H10.4d}$$

$$q_{e2} = c_e \cdot (p_2 - p_0) \tag{H10.4e}$$

where $p_0 = 1.0132 \times 10^5$ N m^{-2} is the ambient air pressure, and $c_e = 0.737 \times 10^{-12}$ kg^{-1} m^4 sec is the external leakage coefficient.

The *load pressure*, p_L, causes a mechanical torque, τ_m, on the motor block, which makes the motor spin, ω_m, and move forward. Thereby an *induced flow*, q_{ind}, is generated. In the process, some of the hydraulic power, $p_L \cdot q_{ind}$, is converted into mechanical power, $\tau_m \cdot \omega_m$. The equations of transformation can be written as:

$$\tau_m = \psi_m \cdot p_L \tag{H10.4f}$$

$$q_{ind} = \psi_m \cdot \omega_m \tag{H10.4g}$$

where $\psi_m = 0.575 \times 10^{-5}$ m^3.

On the mechanical side, the motor experiences inertia and friction. Newton's law can be formulated as follows:

$$J_m \cdot \dot{\omega}_m = \tau_m - \rho_m \cdot \omega_m \tag{H10.4h}$$

where $J_m = 0.08$ kg m^2 is the inertia of the motor block, and $\rho_m = 1.5$ kg m^2 sec^{-1} is the friction constant of the motor.

The load flows, q_{L1} and q_{L2}, in and out of the hydraulic motor are controlled by the four–way servo valve shown in Fig. H10.4b.

The tongue position, x, is normalized such that, $x = 1$ corresponds to the orifices of the valve being entirely open. In the central position, $x = 0$, all four orifices are 5% open, i.e., the servo valve has an underlap of $x_0 = 0.05$.

The flows through the orifices are turbulent. Consequently, they observe a square–root law, as shown in Fig. H10.4c.

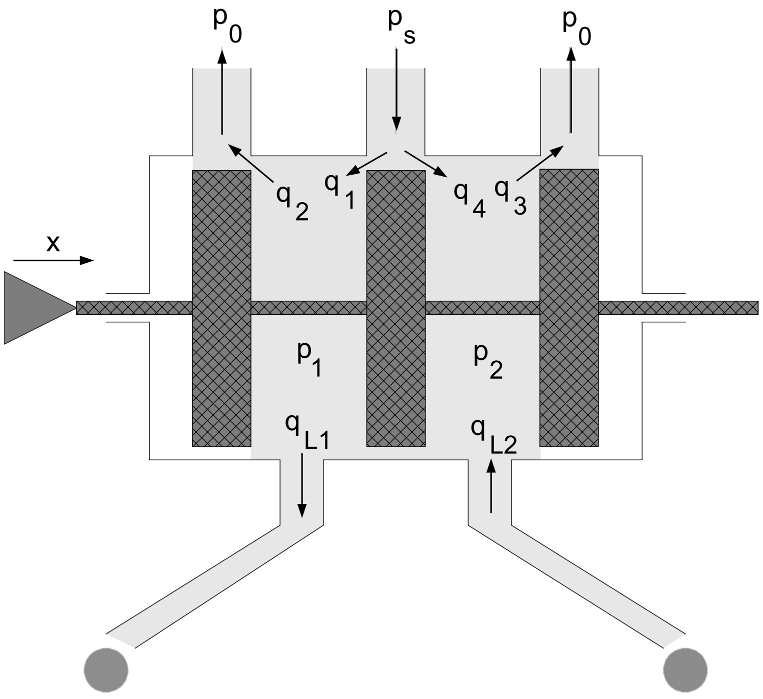

FIGURE H10.4b. Four–way servo valve schematic.

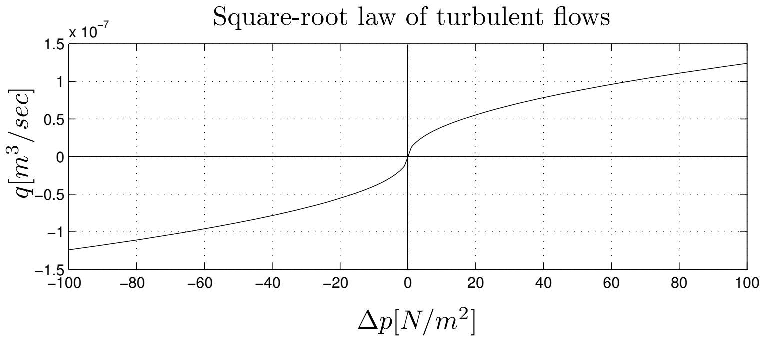

FIGURE H10.4c. Square–root law of turbulent flows of a liquid substance through a narrow orifice.

Thus, each of the four turbulent flows, $q_1 \ldots q_4$, satisfies an equation of the type:

$$q = k \cdot \Delta x \cdot \text{sign}(\Delta p) \cdot \sqrt{|\Delta p|} \tag{H10.4i}$$

where $k = 0.248 \times 10^{-6}$ kg$^{-1/2}$ m$^{7/2}$, Δx is the relative opening of the orifice, i.e. $\Delta x = x_0 \pm x$ limited between zero and one, and Δp is the pressure drop across the orifice. $p_s = 0.137 \times 10^8$ N m^{-2} is the *line pressure* of the hydraulic motor.

¿From Fig. H10.4b, we conclude that:

$$q_{L1} = q_1 - q_2 \tag{H10.4j}$$

$$q_{L2} = q_3 - q_4 \tag{H10.4k}$$

The tongue of the valve is moved by the servo, an electro–mechanical device, depicted in Fig. H10.4d.

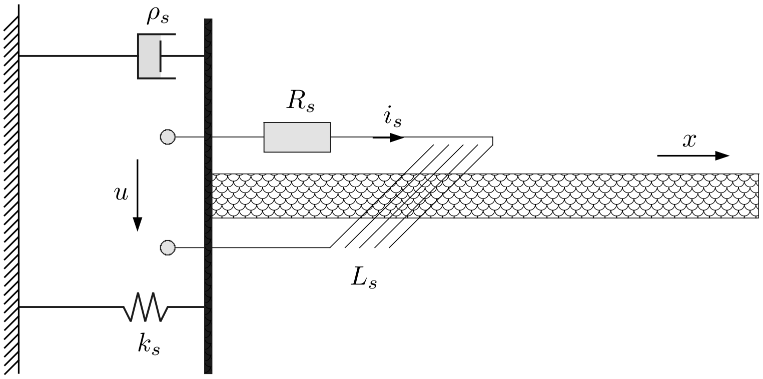

FIGURE H10.4d. Servo schematic.

On the electrical side, the applied voltage, u, causes a current, i_s to flow through a coil. The coil exhibits a resistance, R_s, and an inductance, L_s. Thus:

$$u = R_s \cdot i_s + L_s \cdot \frac{di_s}{dt} + u_{ind} \tag{H10.4l}$$

where $R_s = 1.25 \times 10^{-5}\ \Omega$, and $L_s = 10^{-9}$ H are the normalized resistance and inductance of the coil. The current i_s causes a force, F_s, in the tongue, which makes the tongue move. The velocity of the tongue, v, causes an induced voltage, u_{ind} on the electrical side. In the process, some of the electrical power, $u_{ind} \cdot i_s$, is converted to mechanical power, $F_s \cdot v$. The equations of transformation are:

$$F_s = \psi_s \cdot i_s \tag{H10.4m}$$

$$u_{ind} = \psi_s \cdot v \tag{H10.4n}$$

where $\psi_s = 0.005$ Volts m^{-1} sec $= 0.005$ N Amps^{-1}.

On the mechanical side, the motion of the tongue is opposed by a spring and a damper. Thus, Newton's law can be formulated as follows:

$$m_s \cdot \dot{v} = F_s - k_s \cdot x - \rho_s \cdot v \tag{H10.4o}$$

where $m_s = 0.01$ kg is the normalized mass, $k_s = 400$ N m^{-1} is the normalized spring constant, and $\rho_s = 2$ N m^{-1} sec is the normalized damper coefficient.

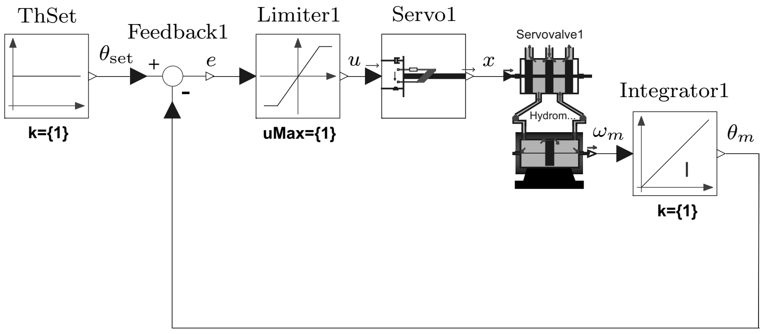

FIGURE H10.4e. Hydraulic motor position control scheme.

The overall position control system is depicted in Fig. H10.4e.

We apply a step of $\theta_{\text{set}} = 1$ rad, and want to observe the step response θ_m as a function of time. The limiter block limits the control signal, u to ± 1. It has also built in a gain factor of $k_l = 0.5$.

This is a 7th–order model with the state vector:

$$\mathbf{x} = (\theta_m, \omega_m, p_1, p_2, x, v, i_s)^T \tag{H10.4p}$$

We simulate the model until $t_f = 0.5$ sec. You can set the initial values of all state variables equal to zero, except for the two pressures, p_1 and p_2, which you should set initially both equal to the arithmetic mean value between the line pressure, p_s, and the ambient air pressure, p_0.

As we wish to prepare this model for real–time simulation, we choose to simulate the model using a fixed–step FE algorithm.

Determine experimentally the largest step size, $h_{\max}$, for which the simulation remains numerically stable. Reduce the step size until the solution is sufficiently accurate. As a criterion for accuracy, we shall compare solutions $\theta_m(t)$ found once with the step size h and once with the step size $h/2$. When the two solutions no longer vary by more than 0.1%:

$$err = \max_{\forall t}(|\theta_m(t)_{[h]} - \theta_m(t)_{[h/2]}|) \leq 0.001 \tag{H10.4q}$$

we consider the solution sufficiently accurate. Find h_{acc}, the step size that produces an accurate solution, and plot the output variable, θ_m as a function of time.

[H10.5] Algebraic Differentiation

For the hydraulic motor of Hw.[H10.4], compute symbolically the Jacobian of the model. Determine its eigenvalues for the initial state, and explain, on the basis of these eigenvalues, the largest step size, $h_{\max}$ found experimentally in Hw.[H10.4].

What can you conclude about the eigenvalue distribution?

[H10.6] Multi–rate Integration

We shall once more consider the hydraulic motor problem of Hw.[H10.4]. We noticed in Hw.[H10.5] that the electrical time constant of the servo valve is faster than the mechanical and hydraulic time constants by several orders of magnitude.

If the step size is indeed dictated by accuracy considerations, we may simply be out of luck. Yet, we may not really require an accuracy of 0.1%. Let us assume that an accuracy of 1% is acceptable. In that case, the step size is limited by the numerical stability rather than by accuracy considerations.

We now wish to implement a multi–rate integration scheme. We keep the step size of the electrical time constant at the maximum level determined earlier, $h_{\max}$, and we increase the step sizes of the other six integrators by making them multiples of $h_{\max}$.

We shall use the single–rate simulation as a reference solution. Determine experimentally, how many time less frequently you may sample the other six integrators, until the multi–rate solution starts differing from the reference solution by more than 1%.

[H10.7] Mixed–mode Integration

We wish to look at the hydraulic motor problem of Hw.[H10.4] once again. This time, we shall replace the FE algorithm of the electrical inductor by the semi–implicit version of the BE algorithm.

Using the same technique proposed earlier to compare the solution computed for step size h with that computed for step size $h/2$, determine the largest step size, h_{acc}, using a mixed–mode integration approach that will offer an accuracy of 1%.

[H10.8] Deep–sea Oil Drilling

We wish to study a deep–sea oil drilling operation. Figure H10.8a shows a deep–sea oil drilling platform with a pipe hanging from it.

The problem was taken from Eitelberg's Ph.D. dissertation [10.14]. The pipe has a length of $\ell = 5$ km. It has a diameter of $\phi = 0.5$ m. The pipe experiences forces from the sea. The inputs, $u(z)$, represent the forces per meter of exposed pipe at a given depth, z.

In accordance with [10.14], the displacement of the pipe, $x(t,z)$, can be modeled as follows:

$$\begin{aligned}\frac{\partial^2 x}{\partial t^2} =& 2 \cdot \frac{\mu_F \cdot v_F}{\mu} \cdot \frac{\partial^2 x}{\partial z \partial t} - \frac{\beta_1}{\mu} \cdot \frac{\partial x}{\partial t} - \frac{\alpha}{\mu} \cdot \frac{\partial^4 x}{\partial z^4} + \left(\gamma(z) - \frac{\mu_f \cdot v_F^2}{2\mu}\right) \cdot \frac{\partial^2 x}{\partial z^2} \\ & - \frac{\bar{\mu}_R \cdot g}{\mu} \cdot \frac{\partial x}{\partial z} + \frac{1}{\mu} \cdot u(z) \qquad \text{(H10.8a)}\end{aligned}$$

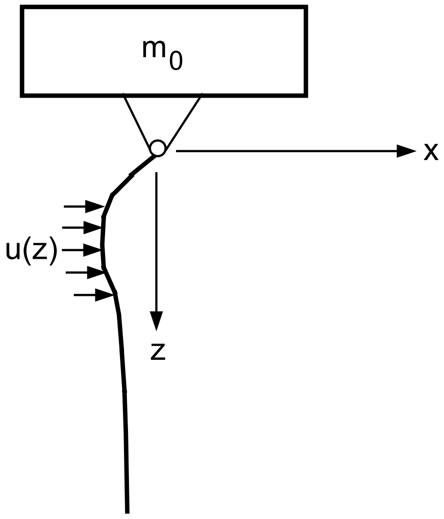

FIGURE H10.8a. Deep–sea oil drilling platform with pipe.

with the abbreviations:

$$\gamma(z) = \frac{g}{\mu} \cdot (m_L - \bar{\mu}_R \cdot (\ell - z)) \tag{H10.8b}$$

$$\mu = \mu_R + \mu_F \tag{H10.8c}$$

where $\mu_R = 173$ kg m^{-1} is the specific mass of the pipe, $\bar{\mu}_R = 150$ kg m^{-1} is a reduced specific mass of the pipe, $\mu_F = 180$ kg m^{-1} is the specific mass of the oil in the pipe, $\alpha = 142 \times 10^6$ kg m^3 sec^{-2} is the torsion stiffness, $\beta_1 = 20$ kg m^{-1} sec^{-1} is the damping coefficient, $v_F = 5$ m sec^{-1} is the velocity of the oil in the pipe, and $m_L = 10^4$ kg is the mass of the weight at the lower end of the pipe.

The boundary conditions can be specified as follows:

$$x\Big|_{z=0} = 0 \tag{H10.8d}$$

$$\left.\frac{\partial^2 x}{\partial z^2}\right|_{z=0} = 0 \tag{H10.8e}$$

$$\left.\frac{\partial^2 x}{\partial z^2}\right|_{z=\ell} = 0 \tag{H10.8f}$$

$$\left.\frac{\partial^2 x}{\partial t^2}\right|_{z=\ell} = \frac{\alpha}{m_L} \cdot \frac{\partial^3 x}{\partial z^3} - g \cdot \frac{\partial x}{\partial z} + \frac{1}{m_L} \cdot u_L \tag{H10.8g}$$

where u_L is the force tugging at the weight.

We shall convert this hyperbolic partial differential equation to a set of ordinary differential equations using the Method–of–Lines approach. To this end, we shall cut the pipe into 50 segments of $\Delta x = 100$ m length each. Thus, we end up with 100 first–order differential equations.

We shall use 4th–order accurate central differences for the first spatial derivatives, 5th–order accurate central differences for the second spatial derivatives, and 5th–order accurate central differences for the fourth spatial derivatives. Towards the boundary, we shall need to use 4th–order accurate biased formulae for the spatial derivatives.

For the second boundary condition at the lower end, we shall use 4th–order accurate biased formulae for the first and third spatial derivatives.

Why did we choose such high–order formulae? We really didn't have any choice. The fourth spatial derivative cannot be discretized, unless we use at least a formula that is 4th–order accurate. Since we use central differences, we get one additional order of accuracy for free. Since we have to use high–order for the discretization of the fourth spatial derivative, we might just as well do the same with the lower–order spatial derivatives, as we get these approximations for free.

Find the Jacobian of the resulting ODE system, and plot its eigenvalues in the complex λ–plane.

[H10.9] Inline Integration

We shall once more consider the oil–drilling operation of Hw.[H10.8]. Since we now know that the eigenvalues are spread up and down, a little to the left of the imaginary axis, we choose the F–stable trapezoidal rule for integration.

Inline the trapezoidal rule, and determine a suitable set of tearing variables. How many tearing variables do you need?

[H10.10] Inline Integration

We shall once more consider the oil–drilling operation of Hw.[H10.8]. this time around, rather than simulating the model using the trapezoidal rule of Hw.[H10.9], we shall inline the Newmark integration algorithm, introduced in Chapter 5 of this book:

$$\mathbf{v_{k+1}} = \mathbf{v_k} + h \cdot [(1-\gamma) \cdot \mathbf{a_k} + \gamma \cdot \mathbf{a_{k+1}}] \tag{H10.10a}$$

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot \mathbf{v_k} + \frac{h^2}{2} \cdot [(1-2\beta) \cdot \mathbf{a_k} + 2\beta \cdot \mathbf{a_{k+1}}] \tag{H10.10b}$$

where $\mathbf{x_k}$, $\mathbf{v_k}$, and $\mathbf{a_k}$ are approximations to the positions, velocities, and accelerations at time step k. We shall implement the method with the parameter values $\beta = 1/4$, and $\gamma = 1/2$.

How many tearing variables do you need now?

10.15 Projects

[P10.1] Helicopter Control

According to Kailath [10.25], the flight of a helicopter near hover conditions can be described by the linear model:

$$\begin{aligned} \dot{\mathbf{x}} &= \begin{pmatrix} -0.02 & -1.4 & 9.8 \\ -0.01 & -0.4 & 0 \\ 0 & 1 & 0 \end{pmatrix} \cdot \mathbf{x} + \begin{pmatrix} 0.9 \\ 6.3 \\ 0 \end{pmatrix} \cdot u \\ \mathbf{y} &= \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \cdot \mathbf{x} \end{aligned} \tag{P10.1a}$$

where x_1 is the horizontal velocity, x_2 is the pitch rate, and x_3 is the pitch angle. The input, u, is the rotor tilt angle. The eigenvalues of the helicopter model are located at $\lambda_1 = -0.6565$, and $\lambda_{2,3} = 0.1183 \pm 0.3678 \cdot j$. Thus, the uncontrolled helicopter is unstable.

We wish to design a stabilizing controller using output feedback and a full–order Luenberger observer [10.25]. The control structure is shown in Fig. P10.1a.

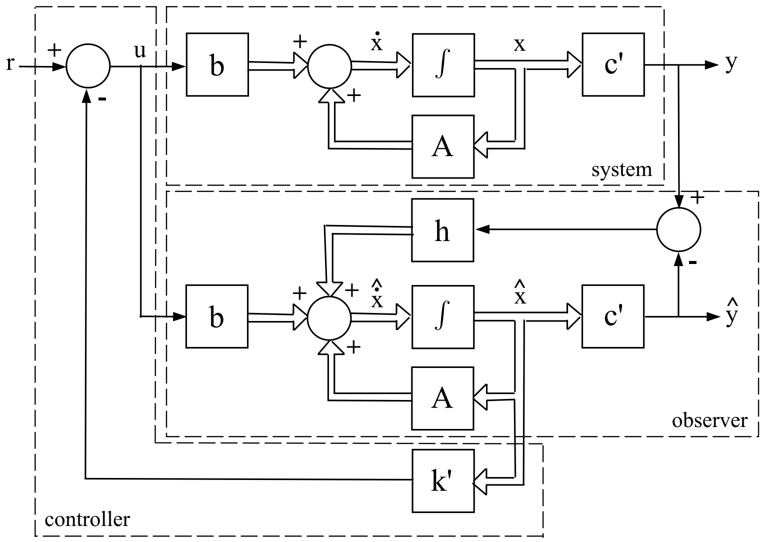

FIGURE P10.1a. Output feedback with full–order Luenberger observer.

The controller works as follows. We can stabilize the helicopter easily by a full state feedback. We can place the poles of the controlled helicopter at $\lambda_{1,2} = -1 \pm j$ and $\lambda_3 = -2$ by multiplying the state vector $\mathbf{x}$ from the left with the vector $\mathbf{k}' = (0.0627, 0.4706, 1)$. The controller gains can be easily computed using any one of a number of *pole placement* algorithms [10.25].

Unfortunately, we don't have the full state available, as only the output variable, y, is being measured. Thus, we run a model of the helicopter in parallel, producing the output, $\hat{y}$. We can obtain a stabilization of the observing model by subtracting the observed output, $\hat{y}$, from the true output, y, and then multiplying that error signal with a vector $\mathbf{h}$, which generates a set of signals that are then fed back to the summing point of the observer model. We can place the observer poles at $\lambda_{1,2} = -2 \pm 2 \cdot j$ and $\lambda_3 = -4$ by choosing $\mathbf{h} = (7.58, 2.2695, 2.4644)^T$.

Since we don't have a helicopter to play around with, we shall create a model of the helicopter that we simulate using a fixed–step RK4 algorithm. We shall implement that model on a computer reading the input signal, u, from an input port, and putting out the output, y, through an output port.

We implement the controller including the observer model on a second computer using an inlined version of the BE algorithm for the observer states. This model reads in the output, y, as input from an input port, and puts out the output, u, through an output port. The second input, r, is implemented in the form of an asynchronous time event, i.e., r remains constant until the user of the system decides to set a new value.

Both computers operate on the same real–time clock. The step size is set such that neither computer experiences overruns.

Real control engineers would go two steps further. They would reduce the order of the observer as much as possible. In the given system, we could get away with a second–order observer. They would then convert the resulting analog controller to an equivalent digital controller designed in the z–domain. We shall not do any of this, as this book is about simulation, and not about control.

We wish to implement the real–time simulation in MATLAB using the HLA architecture. To this end, several hurdles will have to be taken first. The real–time input and output ports are implemented using MATLAB's *Instrument Control Toolbox*, which can communicate with the outside world using a number of different protocols, including TCP/IP.

Yet, this won't solve all of our problems yet. Since MATLAB runs on general–purpose operating systems, such as MS/Windows or Linux, the execution speed of a MATLAB code cannot be guaranteed. None of these operating systems were designed for real–time applications.

However, MATLAB programs (M–files) can be translated to real–time executable C–code using MathWork's *Real-Time Workshop.* This code can then be ported over to a dedicated real–time system using MathWork's *xPC Target* software.

You will need a third computer to implement an HLA kernel that implements the basic RTI functionality and that can communicate with MATLAB's Instrument Control Toolbox.

10.16 Research

[R10.1] Real–time Simulation of Hyperbolic PDEs

We have looked in Hw.[H10.8] at a hyperbolic differential equation that we wished to simulate in real time. It didn't look good at all.

In Chapter 6 of this book, we have learnt that many researchers prefer explicit ODE solvers for dealing with hyperbolic PDEs. However, the FE algorithm won't cut the pie, because it will take incredibly small time steps to capture the eigenvalues close to the imaginary axis inside the stability region of the algorithm. Thus, we would need to use either an AB3 or an RK4 algorithm, which may still be the cheapest solution to the problem.

Multi–rate integration is out of the question, because all of the eigenvalues have similar real parts. For the same reason, also the mixed–mode integration won't work.

We tried inline integration of implicit F–stable algorithms instead, but weren't exactly successful with this approach either. The problem is that we need large numbers of tearing variables, i.e., the Hessian matrix in the Newton iteration is still unacceptably large.

This leads to an open research question: Can integration algorithms be found that are either F–stable of stiffly stable that would allow us to get away with a much smaller number of tearing variables?

Each explicit integrator breaks some loops, thereby reducing the size of the remaining Hessian matrix. Can we selectively turn some of the integrators into explicit integrators, while preserving the overall F–stable or stiffly–stable nature of the algorithm? Could we, for example, find an algorithm that is F–stable or stiffly–stable that integrates all of the velocities, $\mathbf{v_k}$, using an explicit algorithm, while all positions, $\mathbf{x_k}$, are being integrated using an implicit algorithm?

Eitelberg, in his dissertation, went a different route. First, he used non–equidistantly spaced intervals in his discretization scheme, making the intervals more narrow, where the pipe is bent the most, in order to reduce the number of segments needed for a sufficiently accurate representation of the pipe. He then used fifth–order accurate splines for the interpolation. In this way, Eitelberg ended up with a 40^{th}–order model instead of a 100^{th}–order model. Second, Eitelberg then studied systematic *model reduction* algorithms to find different models of lower orders that would still represent the most interesting solution, $x(z = \ell)$, accurately enough. In this way, he was able to reduce the order of the pipe model for control purposes down from forty to eight.