1

Introduction, Scope, Definitions

Preview

The purpose of this chapter is to provide a framework for what this book is to cover. Which are the types of questions that it aspires to answer, and what are the kinds of knowledge that you, the reader, can expect to gain by working through the material presented in this book? What are the relations between real physical systems and their mathematical models? What are the characteristics of mathematical descriptions of physical systems? We shall then talk about simulation as a problem solving tool, and finally, we shall offer a classification of the basic characteristics of simulation software systems.

1.1 Modeling and Simulation: A Circuit Example

Let us begin by modeling a simple electrical circuit. The *circuit diagram* of this circuit is provided in Fig.1.1.

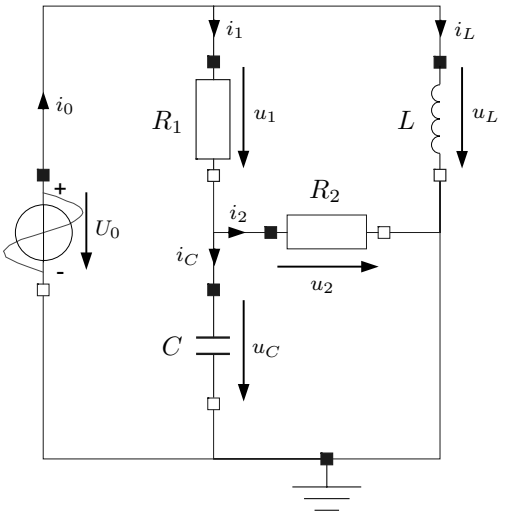

FIGURE 1.1. Circuit diagram of electrical RLC circuit.

Figure 1.1 was produced using *Dymola* [1.11, 1.13], which is currently the most advanced among all of the commercially available physical system modeling and simulation environments. The circuit diagram of Fig.1.1 *is* a mathematical model that can be used to simulate the circuit. It was composed by dragging icons from the graphical electrical component library into the graphical modeling window, dropping them there, and interconnecting them graphically. Associated with each of the icons is the mathematical description of the properties of that particular component model.

The diagram was then edited using a graphical editor to remove the numerical values of the components, and to add names and directions for all currents and voltages. Dymola creates its own names and direction conventions, but does not show them on the circuit diagram using the standard graphical electrical circuit library (this could be changed easily by modifying the component definitions in the library accordingly).

What does Dymola do with the *graphical model* of the circuit? The model is first captured in an alphanumerical form using a modeling language called *Modelica* [1.21]. In the process of compiling the model, the Dymola model compiler performs a lot of symbolic preprocessing on the original mathematical representation. We shall learn more about the symbolic formulae manipulation algorithms that Dymola employs in later chapters of this book. Once a suitable *simulation model* has been derived, it is translated into a C program that then gets compiled further. The compiled model is then simulated by making calls to the numerical run–time library that forms part of the overall Dymola modeling and simulation environment.

What if we were to use a professional circuit simulator, such as *PSpice* [1.19], instead of the more general Dymola software? Modern versions of Spice also offer a *Graphical user Interface (GUI)*, usually called a *schematic capture* program [1.17] in the context of circuit simulation. In the case of Spice, the circuit diagram is captured alphanumerically in the form of a so–called *netlist*. In older versions of Spice, the netlist constituted the user interface, just like older versions of Dymola used a language similar to Modelica as the input language for the description of models.

For the given circuit, the netlist could take the following form:

Vin 1 0 **DC** 10Volts
*R*1 1 2 00Ohms
*R*2 2 0 20Ohms
C 2 0 1uF
L 1 0 1.5mH
.END

Spice, contrary to Dymola, performs hardly any symbolic preprocessing. The netlist is parsed at the beginning of the simulation, and the information contained in it is stored internally in a data structure that is then interpreted at run time.

How about using MATLAB [1.15] to simulate this circuit? MATLAB is of particular interest to us, since it is a wonderful language to describe algorithms in, and since this book is all about algorithms, we shall use MATLAB exclusively in this book for the documentation of these algorithms, as well as for the homework problems that accompany each of the chapters.

MATLAB is not geared toward simulation at all. It is a general purpose programming language supporting high–level data structures that are particularly powerful for the description of algorithms. Since MATLAB wasn't designed to support modeling and simulation, the user will have to perform considerably more work manually, before the circuit description can be fed into MATLAB for the purpose of simulation.

As the circuit contains five separate components in five distinct branches of the circuit, the dynamics of this circuits can be described by 10 variables, namely the five voltages across each of the branches, and the five currents flowing through them. Hence we shall need 10 separate and mutually independent equations to describe the model dynamics in terms of these variables.

The 10 equations can be read out of the circuit diagram easily. Five of them are the *constitutive equations* of the circuit components, relating the voltage across and the current through each of the branches to each other:

$$u_0 = 10 \tag{1.1a}$$

$$u_1 - R_1 \cdot i_1 = 0 \tag{1.1b}$$

$$u_2 - R_2 \cdot i_2 = 0 \tag{1.1c}$$

$$i_C - C \cdot \frac{du_C}{dt} = 0 \tag{1.1d}$$

$$u_L - L \cdot \frac{di_L}{dt} = 0 \tag{1.1e}$$

Three additional equations can be obtained by applying *Kirchhoff's Voltage Law (KVL)* to the circuit, which states that the voltages around a mesh must add up to zero. These are therefore often called the *mesh equations*.

$$u_0 - u_1 - u_C = 0 \tag{1.2a}$$

$$u_L - u_1 - u_2 = 0 \tag{1.2b}$$

$$u_C - u_2 = 0 \tag{1.2c}$$

The final two equations can be obtained by applying *Kirchhoff's Current Law (KCL)* to the circuit, which states that the currents flowing into a node must add up to zero. These are therefore often called the *node equations*. One of the node equations is always redundant, i.e., not linearly independent, and must therefore be omitted. It has become customary to omit the

node equation of the ground node. The two remaining node equations can be written as:

$$i_0 - i_1 - i_L = 0 \tag{1.3a}$$
$$i_1 - i_2 - i_C = 0 \tag{1.3b}$$

These 10 equations together form another equivalent mathematical description of the circuit. They consist of a set of implicitly described partly algebraic and partly differential equations. We call this mathematical description an *implicit differential and algebraic equation (DAE) model*.

We can make the model explicit by deciding, which variable to solve for in each of the equations, and by arranging the equations in such a manner that no variable is being used before it has been defined. We call this the process of *horizontally and vertically sorting* the set of equations. In Chapter 7 of this book, you shall learn how equations can be sorted algorithmically. For now, let us simply present one possible solution to the sorting process.

$$u_0 = 10 \tag{1.4a}$$
$$u_2 = u_C \tag{1.4b}$$
$$i_2 = \frac{1}{R_2} \cdot u_2 \tag{1.4c}$$
$$u_1 = u_0 - u_C \tag{1.4d}$$
$$i_1 = \frac{1}{R_1} \cdot u_1 \tag{1.4e}$$
$$u_L = u_1 + u_2 \tag{1.4f}$$
$$i_C = i_1 - i_2 \tag{1.4g}$$
$$\frac{di_L}{dt} = \frac{1}{L} \cdot u_L \tag{1.4h}$$
$$\frac{du_C}{dt} = \frac{1}{C} \cdot i_C \tag{1.4i}$$
$$i_0 = i_1 + i_L \tag{1.4j}$$

In this model, the equal signs have assumed the role of assignments rather than equalities, which was the case with the previous model. Each unknown appears exactly once to the left of the equal sign, and all variables used in the expressions of the right hand sides have been assigned values, before they are being used.

Notice that the variables u_C and i_L are not treated as unknowns. Since they are the outputs of integrators, they are computed by the integration algorithm used in the simulation, and don't need to be computed by the model. Such variables are referred to as *state variables* in the literature.

We are still confronted with a mixture of algebraic and differential equations, but the model has now become explicit. We call this an *explicit DAE model*.

Sometimes, the explicit DAE model is also called *simulation model*, since the traditional simulation languages, such as *ACSL* [1.18], were able to deal with this type of mathematical description directly.

Although MATLAB can deal with simulation models, this is still not the preferred form to be used when simulating linear systems with MATLAB.

We can now plug the explicit equations into each other, substituting the unknowns on the right hand side by the expressions defining these unknowns, until we end up with equations for the variables du_C/dt and di_L/dt, the so–called *state derivatives*, that depend only on the state variables, u_C and i_L, as well as the input variable, u_0. These equations are:

$$\frac{du_C}{dt} = -\frac{R_1 + R_2}{R_1 \cdot R_2 \cdot C} \cdot u_C + \frac{1}{R_1 \cdot C} \cdot u_0 \tag{1.5a}$$

$$\frac{di_L}{dt} = \frac{1}{L} \cdot u_0 \tag{1.5b}$$

We can add one or several *output equations* for those variables that we wish to plot as simulation results. Let i_2 be our output variable. We can obtain an equation for i_2 that depends only on state variables and input variables in the same fashion:

$$i_2 = \frac{1}{R_2} \cdot u_C \tag{1.6}$$

This mathematical representation is called an *explicit ordinary differential equation (ODE) model*. In the control literature, it is usually referred to as the *state–space model*.

If the state–space model is linear, as in the given case, it can be written in a matrix–vector form:

$$\begin{pmatrix} \frac{du_C}{dt} \\ \\ \frac{di_L}{dt} \end{pmatrix} = \begin{pmatrix} -\frac{R_1+R_2}{R_1 \cdot R_2 \cdot C} & 0 \\ \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} u_C \\ \\ i_L \end{pmatrix} + \begin{pmatrix} \frac{1}{R_1 \cdot C} \\ \\ \frac{1}{L} \end{pmatrix} \cdot u_0 \tag{1.7a}$$

$$i_2 = \begin{pmatrix} \frac{1}{R_2} & 0 \end{pmatrix} \cdot \begin{pmatrix} u_C \\ \\ i_L \end{pmatrix} \tag{1.7b}$$

This model finally is in an appropriate form for feeding it into MATLAB. The following MATLAB code may be used to simulate the circuit:

```
% Enter parameter values
%
R1 = 100;
R2 = 20;
L = 0.0015;
C = 1e-6;
%
% Generate system matrices
%
R1C = 1/(R1 * C);
R2C = 1/(R2 * C);
a11 = -(R1C + R2C);
A = [ a11 , 0 ; 0 , 0 ];
b = [ R1C ; 1/L ];
c = [ 1/R2 , 0 ];
d = 0;
%
% Make a system and simulate
%
S = ss(A, b, c, d);
t = [ 0 : 1e-6 : 1e-4 ];
u = 10 * ones(size(t));
x0 = zeros(2, 1);
y = lsim(S, u, t, x0);
%
% Plot the results
%
subplot(2, 1, 1)
plot(t, y, 'k-')
grid on
title('\tex{Electrical RLC Circuit}')
xlabel('\tex{time}')
ylabel('\tex{$i_2$}')
print -deps fig1_2.eps
return
```

The simulation results are presented in Fig.1.2.

Clearly, MATLAB employs a considerably lower–level user interface than either Dymola or PSpice, but maybe that is good, since the purpose of this book is to teach simulation methods.

Do we now understand more about how the simulation was performed using MATLAB? Unfortunately, this question must be answered in the negative. The entire simulation takes place inside the *lsim* box, which we haven't opened yet. The main purpose of this book is to open up the *lsim* box, and understand, how it has been built, but more about that later.

To be fair to MATLAB, it must be mentioned that also MATLAB, just like its competitors, offers a graphical user interface, called *SIMULINK* [1.7]. However, SIMULINK is not a schematic capture program. It is only a *block diagram editor*.

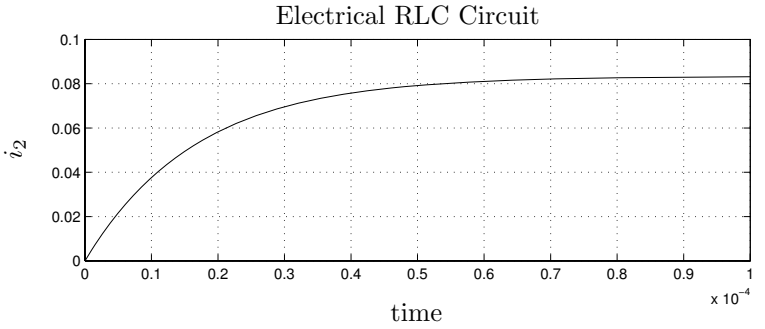

FIGURE 1.2. Simulation results of electrical RLC circuit.

SIMULINK is thus located at the level of the explicit DAE model. Given that model, we can start by drawing the two integrator boxes, and then work ourselves backward toward the input variable, and forward toward the output variable. The resulting block diagram is shown in Fig.1.3.

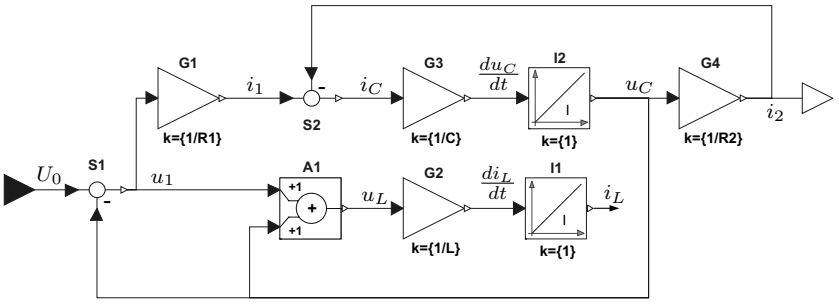

FIGURE 1.3. Block diagram of electrical RLC circuit.

Figure 1.3 was not drawn using SIMULINK, but instead, we chose to draw the figure in Dymola, using Dymola's graphical block diagram library. We then edited the graph manually by adding the names of the variables to each of the signals.

What was gained by representing the explicit DAE model graphically as a block diagram? The only advantage of doing so is that it becomes evident from the block diagram that the integrator computing the variable i_L could have been pruned away, as it does not contribute at all to computing the output variable.

Block diagrams are useful tools for representing control systems. They are not useful, however, for representing electrical circuits.

1.2 Modeling vs. Simulation

In the previous section, we have shown a full modeling and simulation cycle, starting out with a physical system, an electrical RLC circuit, and ending with the display of the trajectory behavior of the output variable, i_2.

The process of *modeling* concerns itself with the extraction of knowledge from the physical plant to be simulated, organizing that knowledge appropriately, and representing it in some unambiguous fashion. We call the end product of the modeling cycle the *model* of the system to be simulated.

The process of *simulation* concerns itself with performing experiments on the model to make predictions about how the real system would behave if these very same experiments were performed on it.

At the University of Arizona, we offer currently two senior/graduate level classes dealing with the issues of modeling and simulating physical systems. One of them, *Continuous System Modeling*, deals with the issues of creating suitable models of physical systems. For it, the companion book of this textbook, also entitled *Continuous System Modeling* [1.6], was developed. The other, *Continuous System Simulation*, concerns itself with the issues of simulating these models accurately and efficiently. For that class, this textbook has been written.

A question that you, the reader, may already have begun to ask yourself is the following: Where does modeling end and simulation begin?

In the old days, we might have answered that question in the following way: Simulation is what is being done by the computer, whereas modeling concerns the steps that the modeler has to undertake manually in order to prepare the simulation program.

Yet, this answer is not very satisfactory. We have seen that, when using MATLAB to simulate the circuit, the modeler had to do much more manual preprocessing than when using either Dymola or Spice. The answer to the above question would thus depend on the simulation tool that is being used. This is not very useful.

A more gratifying answer may be obtained by looking at Fig.1.4.

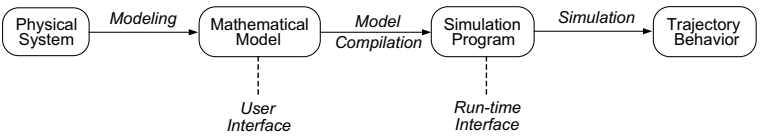

FIGURE 1.4. Modeling *vs.* simulation.

The whole purpose of the *mathematical model* is to provide the human user of the modeling and simulation environment with a means to represent knowledge about the physical system to be simulated in a way that is as *convenient* to him or her as possible. Modeling is thus indeed always done manually. The mathematical model represents the user interface. It has

absolutely nothing to do with considerations of how that model is going to be used by the simulation engine.

Which is the most appropriate mathematical model of a system to be simulated depends on the nature of the physical system itself, and maybe also on the types of experiments that are to be performed on the model.

We have already mentioned that a *block diagram* may be a suitable tool to represent the knowledge needed to simulate a control system. It is certainly not a convenient tool to represent the knowledge needed to simulate an electrical circuit. A *circuit diagram*, on the other hand, may be the most natural way to represent an electrical circuit, as long as the experiment to be performed on the model does not concern itself with non–electrical phenomena, such as the heating of the device that results from current flowing through resistors, and the temperature dissipation of the package, in which the circuit has been integrated. In that case, a *bond graph* may be a much better choice for representing the physical knowledge needed to simulate the circuit.

The bond graph of the above circuit is shown in Fig.1.5.

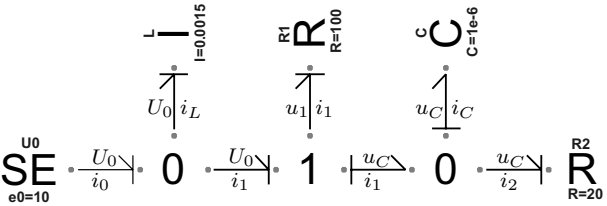

FIGURE 1.5. Bond graph of electrical RLC circuit.

Figure 1.5 was produced using Dymola's graphical bond graph library [1.3]. Bond graphs play an important role in the companion book, *Continuous System Modeling* [1.6], to this text. They are of no concern to this class, since they are only used to the left of the mathematical model in Fig.1.4. In this textbook, we do not concern ourselves with issues to the left of the mathematical model.

Once the mathematical model has been formulated, the modeling and simulation environment can make use of that model to perform simulations, and produce simulation results. For models as simple as our electrical circuit, either of the three representations: the circuit diagram, the block diagram, or the bond graph, can be simulated equally easily, accurately, and efficiently. The user simply instructs Dymola to simulate the model, Dymola then performs the necessary model compilations, executes the simulation run, and prepares the variables in a data base, such that the user can then pick the output variable(s) he or she is interested in, and plot them.

Since modeling of a physical system is always done manually, it is evi-

dent that we need to offer a class, teaching the students, how to generate a model of a physical system that is suitable for performing a given set of experiments on it. Yet, if everything to the right of the mathematical model in Fig.1.4 can be fully automated, why should an engineering student concern him- or herself at all with simulation issues? Why not leave these issues to the experts, i.e., the applied mathematicians?

Unfortunately, things are not going always as smoothly as in this simple electrical RLC circuit. It happens more often than not that a simulation does not produce the desired results the first time around. A user who only understands modeling and uses the simulation environment as a black box will most likely be at a total loss as to what went wrong and why, and he or she will have no inkling as to how the problems can be overcome. In fact, the more complex the symbolic formulae manipulation algorithms are that are being employed by the modeling and simulation environment as part of the model compilation, the less likely it is that an uninformed user of that environment will be able to make sense out of error messages that result from mishaps happening at run time, the so–called *run–time exceptions*.

The main purpose of this class and this textbook are to prepare the student for anything that the modeling and simulation environment may throw at him or her. The knowledge provided in this textbook will enable the simulation practitioner to deal with all eventualities that he or she may come across in the adventure of simulating a mathematical model effectively and efficiently.

Let us return once more to Fig.1.4. What does the other interface, the run–time interface, represent? The purpose of that interface is to define a simulation model that can be simulated *efficiently* and *accurately*.

It was already mentioned that Spice essentially simulates the netlist directly, whereas Dymola performs a lot of symbolic preprocessing on the model, i.e., the distance between the mathematical model and the simulation program is very small in Spice, whereas it is impressively wide in Dymola.

You shall learn in this class that it actually matters, which way we proceed. The algorithms underlying Spice simulations only work because the possible structures of an electronic circuit are very well defined and don't change much from one circuit to the next. On the other hand, if we were to simulate how a circuit heats up during simulation, and simultaneously wanted to simulate how the electrical parameter values (the resistances and capacitances) change in function of the current device temperature, the algorithms underlying the Spice simulation, the so–called *sparse tableau equations* that are used in a *modified nodal analysis*, would break down, because the so modified model would contain additional algebraic loops that these algorithms could not possibly handle.

Thus, the most appropriate run–time interface is also a function of the system to be simulated, and possibly of the experiment or set of experiments to be performed on the model. Yet, this interface only concerns itself

with the way, the simulation algorithms work. It has no bearing whatsoever on how the user represents his or her mathematical model.

In which book are the model compilation issues to be discussed? Since both interfaces move around, i.e., they are sometimes a little further to the right, and sometimes a little further to the left, it is important to look at these issues both from the perspective of a modeler and from that of a simulation practitioner. Hence there is a certain degree of overlap and redundance between the two textbooks as far as model compilation algorithms are concerned. This decision was taken on purpose to allow the students to take the two classes in any sequence. Neither of them depends on the knowledge provided in the other.

1.3 Time and Again

In the real world, time simply happens. We can measure it, but we cannot influence it. Every morning, when we wake up, we have aged by precisely one day since the previous morning. There is nothing to be done about. If we are slow, in getting something done, we have to hurry up, as we cannot slow time down.

In simulation, time does not simply happen. We need to make it happen. When we simulate a system, it is our duty to manage the simulation clock, and how effectively we are able to manage the simulation clock will ultimately decide upon the efficiency of our simulation run.

In the previous two sections of this chapter, we have looked at different ways for representing a model. At the bottom of the hierarchy, we encountered the *explicit ODE model*, which we also called the *state–space model*. We simulated a simple electrical RLC circuit, represented as a linear state–space model, by use of MATLAB, and obtained a trajectory for the output variable, i_2, as a function of time. That output trajectory was depicted graphically in Fig.1.2.

The trajectory $i_2(t)$ seems to be a real–valued function of one real–valued argument. For any value of t, we can obtain the appropriate value of i_2. Yet, this is only an illusion, created to make us believe that the simulation is a faithful image of how we perceive the real system to work.

A digital computer has no means of computing numerically any real–valued function of a real–valued argument. To do so would require an infinite amount of real time. Instead, the time axis in the simulation must be *discretized*, such that the total number of discrete time points within the range of simulated time remains finite, and the simulation must proceed by jumping from one discrete time point to the next. The coarser we can choose the discretization in time, the smaller the total number of discrete time points will be, and consequently, the less work needs to be done in the simulation to evaluate the model at the output points. The discretization

in time directly influences the efficiency of the simulation run.

Consequently, neither of the previously introduced model types can be simulated directly. Inside the simulation box, the model gets converted once more by reducing differential equation models to *difference equation (ΔE) models*. Thereby, an explicit ODE model is converted to an explicit ΔE model, whereas an implicit DAE model is converted to an implicit ΔAE model, etc.

The illusion of a continuous $i_2(t)$ curve was created by making the plot routine connect neighboring data points using a straight–line approximation. How often do we need to actually compute values of i_2? We need to do so sufficiently often that the straight–line approximation looks smooth to the naked eye. We call the distance between two neighboring computed output data points the *communication interval*. When we simulate a system, the simulation software asks us to provide that information to it. In the MATLAB code, we created a vector:

$t = [\ 0\ :\ 1\text{e-}6\ :\ 1\text{e-}4\ ];$

of communication points. It states that we wish to compute the output variable once every 10^{-6} seconds up until the final time of 10^{-4} seconds, giving us a result vector of 101 data points.

Does this mean that the simulation proceeds at the pace dictated by the communication grid? Absolutely not. The communication grid was only created to please the user, such that he or she can enjoy the illusion of a smoothly looking output variable. The simulation pace, however, is dictated by the numerical needs of the algorithm. The more accurately we wish to simulate, the smaller the time steps of the simulation must be chosen.

Thus, the simulation clock can advance either more slowly than the communication clock by allowing multiple simulation steps to occur within a single communication interval, or it could proceed more rapidly. In the latter case, the intermediate output points are obtained not by *simulation*, but by *interpolation*. If the interpolation routine can produce an interpolation of the same order of approximation accuracy as the integration, this is a perfectly valid way of computing output points.

Figure 1.6 depicts the relationship between the different types of *time* that we have to deal with in a simulation.

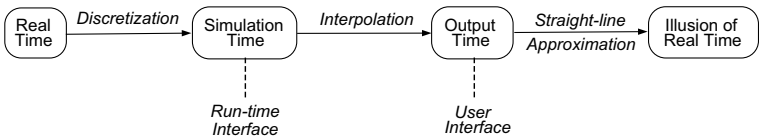

FIGURE 1.6. The different faces of *time*.

Whereas the communication grid is usually equidistantly spaced, the simulation grid is not. The step size, h, of the simulation is usually allowed

to adjust itself, such that the accuracy requirements are met. A simulation user knows how to set the *communication interval* or *sampling rate*, t_s, but he or she wouldn't know how to set the step size, h, of the simulation. Consequently, most simulation software systems will ask the user to specify an *accuracy requirement* instead. The integration algorithm uses some formula to estimate the numerical *integration error*, and then uses a control scheme to adjust the step size such that the integration error is kept as large as possible, while not exceeding the specified maximum error.

Does the simulation clock at least advance monotonously with real time, i.e., will the time difference, Δt, of the simulation clock between two subsequent evaluations of the model be always positive? Unfortunately, also this question must usually be answered in the negative for three separate reasons.

1. The step size, h, is not necessarily identical with the time advance, Δt, of model evaluations. Many integration algorithms, such as the famous Runge–Kutta algorithms, which we shall meet in Chapter 3 of this book, perform multiple model evaluations within a single time step. Thus, each time step, h, contains several micro–steps, Δt, whereby Δt is not a fixed divider of h. Instead, the simulation clock may jump back and forth within each individual time step.

2. Even if the integration algorithm used is such that Δt remains positive at all times, the simulation clock does not necessarily advance monotonously with real time. There are two types of error–controlled integration algorithms that differ in the way they handle steps that exhibit an error estimate that is too large. *Optimistic algorithms* simply continue, in spite of the exceeded error tolerance, while reducing the step size for the subsequent step. In contrast, *conservative algorithms* reject the step, and repeat it with a smaller step size. Thus, whenever a step is rejected, the simulation clock in a conservative algorithm turns back to repeat the step, while not committing the same error. Wouldn't it be nice if we could do the same in the real world?

3. Even if an optimistic algorithm with positive Δt values is being employed, the simulation clock may still not advance monotonously with real time. The reason is that integration algorithms cannot integrate across discontinuities in the model. Thus, if a discontinuity is encountered somewhere inside an integration step, the step size must be reduced and the step must be repeated, in order to place the discontinuity in between subsequent steps. These issues shall be discussed in Chapter 9 of this book.

Hence the flow chart shown in Fig.1.6 is still somewhat oversimplified, as it does not account for the micro–management of time within a single integration step.

The issues surrounding *time management* as part of the simulation algorithms shall haunt us throughout the various chapters of this book.

1.4 Simulation as a Problem Solving Tool

Simulation has become the major analysis tool in essentially all of engineering, and much of science. Industry nowadays demands that companies providing parts for their products ship their parts with simulation models that can be assembled in just about the same fashion as the real system is. For example, when you buy these days an all–American car, you may not want to check too closely what is under the hood, because you may quickly discover that your car comes equipped with a German engine and a Japanese transmission.

Car manufacturers these days allow two years from the conception of a new model, until the first cars roll off the production line. During the first year, the car itself is designed and its performance is optimized by means of continuous system simulation; during the second year, the production process of the car is designed, again involving a lot of simulation, though mostly of a discrete event nature.

This can only work if the parts come equipped with ready–to–use simulation models that can be plugged together quickly and painlessly. This is only possible if the modeling methodology in use is *object oriented*, which invariably leads to large sets of implicitly defined DAE systems.

To this end, the *Modeling and Simulation (M&S) environment* must be able to deal with implicit DAE descriptions, either by simulating such descriptions directly, or by automatically converting them to explicit ODE descriptions beforehand. The days of 10,000 lines of spaghetti FORTRAN code to e.g. simulate the flight of a missile, taking into account such gory details as the seeker and its gyroscopically stabilized platform, as well as the flopping around of the liquid fuel in the fuel tank, are thus finally over.

Whereas the issues surrounding object–oriented modeling are not the aim of this book[1], issues surrounding the symbolic model transformations to precondition the simulation code for efficient run–time performance are being dealt with in later chapters of this textbook.

Modern M&S environments, such as Dymola [1.11, 1.13], are capable of automatically generating simulation code from an object–oriented mathematical model that runs as efficiently as, if not more efficiently than, the best among the hand–coded spaghetti simulation programs of the past. The translation of the model is usually accomplished within seconds of real time.

[1]These issues are discussed both extensively and intensively in the companion book of this text, *Continuous System Modeling* [1.6].

In the past, the life cycle of a simulation program often extended beyond that of its designer. The engineer who originally designed and wrote the spaghetti simulation code retired before the program itself had reached the end of its usefulness. Maintaining these programs, after the original designer could no longer be consulted, was an absolute nightmare. Also these days are luckily over.

1.5 Simulation Software: Today and Tomorrow

We published an article with the same title, *Simulation Software: Today and Tomorrow*, a little over 20 years ago [1.5], because at that time, we felt that the earlier article discussing similar topics [1.4] had meanwhile outlived its usefulness.

Reading through the 1983 paper once more, we recognize and happily acknowledge how hopelessly outdated that article has meanwhile become. This discovery is a cause of excitement, not depression, because it shows us how incredibly active this research area has been over the past 20 years, and how wonderfully dynamic this research area continues to be to this day.

Although the principles of *object–oriented modeling* had been developed already in the sixties [1.8], *Simula 67* had only been designed for discrete event simulation, not for continuous system simulation, and these concepts could not easily be carried over to modeling physical systems. The reason is that, in discrete event simulation, we always know what are the causes, and which are their effects. In physical system modeling, this is not the case. The *computational causality* of physical laws can therefore not be predetermined, but depends on the particular use of that law. We cannot conclude whether it is the current flowing through a resistor that causes a voltage drop, or whether it is the difference between the potentials at the two ends of the resistor that causes current to flow. Physically, these are simply two concurrent aspects of one and the same physical phenomenon. Computationally, we may have to assume at times one position, and at other times the other.

First attempts at dealing with the problems of physical system modeling in an object–oriented fashion were developed simultaneously in two seminal Ph.D. dissertations By Elmqvist [1.11] and Runge [1.20]. Whereas Elmqvist focused his attention on symbolic formulae manipulation as a tool for preconditioning the model equations to obtain efficiently executing simulation code, Runge attempted to solve implicit DAE models directly.

Whereas a first prototypical implementation of *Dymola* had been implemented by Hilding Elmqvist already as part of his Ph.D. dissertation [1.11], Dymola was not yet capable of dealing with large–scale engineering models in those days. The code got stuck, as soon as it encountered either

an *algebraic loop* or a *structural singularity*, which happened invariably in most large–scale engineering models.

First attempts at tackling the algebraic loop and structural singularity problems in a completely generic fashion were undertaken by Hilding Elmqvist in 1993 [1.2]. This research was followed up in 1994 by an important paper on *symbolic tearing* methods [1.10]. By 1997, heuristic procedures had been developed to automatically identify a suitable set of tearing variables. By that time, we finally had available a tool that could reduce, in a fully automated fashion, any implicit DAE model to explicit ODE form. We shall talk much more about these algorithms in Chapter 7 of this book.

A first prototype of a *Graphical User Interface (GUI)* for Dymola was created by Hilding Elmqvist as early as 1982 [1.12]. The graphical software *HIBLIZ* [1.12] had a number of interesting features, yet it was far ahead of its time, as the computer hardware of those days wasn't ready yet for these types of applications. Elmqvist resumed his work on a GUI for Dymola in 1993, which resulted in a very powerful modern graphical software environment, of which you have already seen some samples earlier in this chapter.

Elmqvist proved to be one of the most innovative and visionary researchers in M&S methodology and technology of the last quarter of a century, and his *Dynamic Modeling Laboratory*, *Dymola*, has become the de facto industry standard by now. No other tool on the market comes even close to Dymola in terms of flexibility and generality of its use.

On the numerical front, progress has been a bit less spectacular. The 4^{th}–order Runge–Kutta algorithms in use today are still the same algorithms that were known and used in 1983. However, the development of production–grade direct DAE solvers [1.1], a direct outflow of Runge's earlier work, fell in this time frame, and stirred quite a bit of excitement among applied mathematicians.

Furthermore, a lot has happened in terms of the development of better software aiding the design of new numerical algorithms. *MATLAB* [1.15] has become the de facto industry standard for the description of numerical algorithms. All of the algorithms described in this book are explained in terms of snippets of MATLAB code, and most of the homework problems are designed to be solved using MATLAB.

In the same context, the quite impressive advances in the development of tools for *computational algebra* deserve to be mentioned as well. Applied mathematicians like to present the coefficients of their algorithms symbolically as rational expressions, rather than numerically as numbers with many digits after the dot, because in this way, the numerical accuracy of the algorithm can fully exploit the available mantissa length of the computer on which the algorithm is being implemented. Tools, such as *MAPLE* [1.16] and *Mathematica* [1.22] have made the design of new algorithms considerably less painful than in the past, and indeed, several errors were recently discovered using computational algebra tools in a number of numerical al-

gorithms that had been around for decades [1.14]. When developing this book, we made frequent use of MATLAB's *symbolic toolbox*, which is based on MAPLE, to derive correct rational expressions for the coefficients of new algorithms.

The advent of ever more powerful computer hardware made it possible to search for new algorithms much more efficiently than in the past. For example, we could not have developed the higher–order stiffly–stable linear multi–step methods that are described in Chapter 4 of this book as little as 10 years ago, since several of the search algorithms used in the process milled for more than 30 minutes of real time on a 2.5 GHz personal computer, whereas 10 years ago, we had to rely on a 1 MHz VAX computer for all of our computations.

Finally, the automatic preconditioning of models by means of symbolic formulae manipulation made it possible to employ highly promising numerical algorithms that could not have been used previously, because they would have forced the users to manually convert the models in a manner, which would have been far too cumbersome for them. A good example of this are the *inline integration* algorithms [1.9] that are discussed in Chapter 8 of this book.

For these reasons, we expect that a good number of exciting new numerical algorithms will appear in the open literature at a much more rapid pace over the next few years.

What are tools that are still missing or unsatisfactory in Dymola? A first issue to be improved is the mechanism, by which run–time exceptions are reported back to the user. Advanced *reverse engineering mechanisms* ought to be put in place to translate run–time exceptions back to terms that are related to the original model, i.e., terms that the user of the Dymola M&S environment can understand. Right now, the debugging of Dymola models can be quite challenging.

A second issue to be looked into concerns Dymola's way of handling table–lookup functions. The treatment of tabular functions is unsatisfactory on several counts.

1. If an input variable is provided to the simulation engine in the form of a table, sampled once per communication interval, Dymola uses linear interpolation to estimate intermediate values of the input variable. Yet, the simulation engine may simulate the model using a higher–order algorithm, possibly subdividing the communication interval into several steps. This situation can be remedied easily by use of the Nordsieck vector approach that is discussed in Chapter 4 of this book.

2. If the independent variable of a table–lookup function is not *time*, but a dependent variable of the model, the situation gets more complicated. Yet, the necessary history information could be traced back also in this case. Furthermore, the effects of reduced–order numerical

approximations of table–lookup functions on the overall simulation accuracy ought to be properly studied. This has not happened to date. This could be a nice research topic for a young aspiring applied mathematician.

3. The treatment of large tables, as currently implemented in Dymola, is highly inefficient. This is a compiler issue that will need to be addressed.

4. Large multi–dimensional tables need to be interpolated directly on the storage medium, rather than loading them into the model, and manipulating them at compile time. This is not currently the case. However, the use of *Modelica* as the underlying alphanumerical model representation helps in this respect. Modelica is a full–fledged language, in which adequate table–lookup mechanisms could easily be implemented [1.21].

A third and very interesting research issue concerns the automated assembly of models. For example, if we wish to model a chemical reaction system, we ought to be able to automatically extract the necessary parameters and table–lookup functions from the open literature.

How do we go about such modeling issues today? We probably would use *Google* to find the missing information on the web. Google has become the de facto standard for finding the answer to pretty much any question that we may have. Google has become our most important interface to the accumulated world knowledge.

Yet in order to use Google effectively, we must first come up with the right keywords to find the most suitable articles on the web, and it will be furthermore our task to manually sift through the articles returned to find what we need.

We foresee the need to automate these two current user interfaces as part of a future *distributed M&S environment*. The M&S environment ought to be able to automatically query a distributed data base for the availability of entire models, model parameter values, and table–lookup functions. This demand could provide challenging and exciting research topics for several Ph.D. students of computer science.

1.6 Summary

In this chapter, we started out with a set of different ways how a mathematical model of a physical system can be formulated. We demonstrated that it is important to distinguish the mathematical model (the user interface) from the simulation program (the run–time interface), such that the mathematical model can be defined to maximize the convenience for the

human user of the tool, whereas the simulation program can be defined to optimize run–time efficiency of the simulation code.

We looked at the important issue of time management during execution of a continuous system simulation program with a bird's eye's view. Whereas all of these issues will be revisited throughout the chapters of this book, we considered it useful to bring these issues to the reader's attention early on.

The chapter ended with a discussion of where we stand today in terms of modeling and simulation environments, and what additional features we expect will be required in the near future.

1.7 References

[1.1] Kathryn E. Brenan, Stephen L. Campbell, and Linda R. Petzold. *Numerical Solution of Initial–Value Problems in Differential–Algebraic Equations*. North–Holland, New York, 1989. 256p.

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

[1.3] François E. Cellier and Robert T. McBride. Object–oriented Modeling of Complex Physical Systems Using the Dymola Bond–graph Library. In François E. Cellier and José J. Granda, editors, *Proceedings of the 2003 SCS Intl. Conf. on Bond Graph Modeling and Simulation*, pages 157–162, Orlando, Fl., 2003. The Society for Modeling and Simulation International.

[1.4] François E. Cellier. Continuous System Simulation by Use of Digital Computers: A State–of–the–Art Survey and Perspectives for Development. In Mohamed H. Hamza, editor, *Proceedings Simulation'75*, pages 18–25, Zurich, Switzerland, 1975. ACTA Press.

[1.5] François E. Cellier. Simulation Software: Today and Tomorrow. In Jacques Burger and Yvon Varny, editors, *Proceedings of the IMACS Symposium on Simulation in Engineering Sciences*, pages 3–19, Nantes, France, 1983. North–Holland Publishing.

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

[1.7] James B. Dabney and Thomas L. Harman. *Mastering SIMULINK 4*. Prentice–Hall, Upper Saddle River, N.J., 2001. 432p.

[1.8] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. Simula 67 Common Base Language. Technical report, Norwegian Computing Center, Oslo, Norway, 1968.

[1.9] 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.

[1.10] 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.

[1.11] Hilding Elmqvist. *A Structured Model Language for Large Continuous Systems*. PhD thesis, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978.

[1.12] Hilding Elmqvist. A Graphical Approach to Documentation and Implementation of Control Systems. In *Proceedings* 3^{rd} *IFAC/IFIP Symposium on Software for Computer Control (SOCOCO'82)*, Madrid, Spain, 1982.

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

[1.14] Walter Gander and Dominik Gruntz. Derivation of Numerical Methods Using Computer Algebra. *SIAM Review*, 41(3):577–593, 1999.

[1.15] Duane Hanselman and Bruce Littlefield. *Mastering MATLAB 6*. Prentice–Hall, Upper Saddle River, N.J., 2001. 832p.

[1.16] André Heck. *Introduction to Maple*. Springer Verlag, New York, 2^{nd} edition, 1996. 525p.

[1.17] Marc E. Herniter. *Schematic Capture with Cadence PSpice*. Prentice–Hall, Upper Saddle River, N.J., 2^{nd} edition, 2002. 656p.

[1.18] Edward E. L. Mitchell and Joseph S. Gauthier. *ACSL: Advanced Continuous Simulation Language — User Guide and Reference Manual*. Mitchell & Gauthier Assoc., Concord, Mass., 1991.

[1.19] Franz Monssen. *OrCAD PSpice with Circuit Analysis*. Prentice–Hall, Upper Saddle River, N.J., 3^{rd} edition, 2001. 400p.

[1.20] Thomas F. Runge. *A Universal Language for Continuous Network Simulation*. PhD thesis, Dept. of Computer Science, University of Illinois, Urbana–Champaign, Ill., 1977.

[1.21] Michael M. Tiller. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, Boston, Mass., 2001. 368p.

[1.22] Stephen Wolfram. *The Mathematica Book*. Wolfram Media, Inc, Champaign, Ill., 5^{th} edition, 2003. 1488p.

1.8 Homework Problems

[H1.1] Different Mathematical Models

Given the electrical circuit shown in Fig.H1.1a.

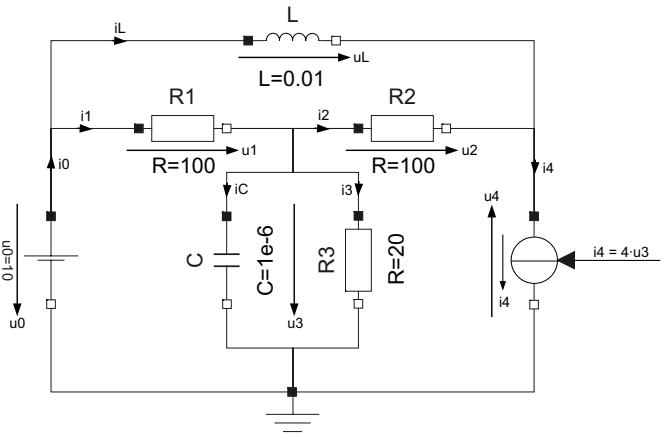

FIGURE H1.1a. Electrical circuit.

The circuit contains a constant voltage source, u_0, and a dependent current source, i_4, that depends on the voltage across the capacitor, C, and the resistor, R_3.

Write down the element equations for the seven circuit elements. Since the voltage u_3 is common to two circuit elements, these equations contain 13 rather than 14 unknowns. Add the voltage equations for the three meshes and the current equations for three of the four nodes. One current equation is redundant. Usually, the current equation for the ground node is therefore omitted.

Formulate an implicit DAE model of this circuit by placing all unknowns to the left of the equal sign, and all known expressions to the right of the equal sign.

Sort the equations both horizontally and vertically. Since you haven't learnt yet a systematic algorithm for doing this (such an algorithm shall be presented in Chapter 7 of this book), use intuition to come up with the sorted set of equations.

Formulate an explicit DAE model of this circuit using the sorted equations.

Use variable substitution to derive a state–space model of this circuit in matrix–vector form. We shall assume that u_3 is our output variable.

Simulate the circuit across 50 μsec using MATLAB's *lsim* function. Store 101 equidistantly spaced output values, and plot the output variable as a function of time.

[H1.2] Discretization of State Equations

Given the following explicit ODE model:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} + \mathbf{b} \cdot u \tag{H1.2a}$$

$$y = \mathbf{c}' \cdot \mathbf{x} + d \cdot u \tag{H1.2b}$$

where:

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -2 & -3 & -4 & -5 \end{pmatrix} \tag{H1.2c}$$

$$\mathbf{b} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{H1.2d}$$

$$\mathbf{c}' = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} \tag{H1.2e}$$

$$d = 10 \tag{H1.2f}$$

Engineers would usually call such a model a *linear single–input, single–output (SISO) continuous–time state–space model*.

We wish to simulate this model using the following integration algorithm:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot \dot{\mathbf{x}}_\mathbf{k} \tag{H1.2g}$$

which is known as the *Forward Euler (FE) integration algorithm*. If $\mathbf{x_k}$ denotes the state vector at time t^*:

$$\mathbf{x_k} = \mathbf{x}(t)\Big|_{t=t^*} \tag{H1.2h}$$

then $\mathbf{x_{k+1}}$ represents the state vector one time step later:

$$\mathbf{x_{k+1}} = \mathbf{x}(t)\Big|_{t=t^*+h} \tag{H1.2i}$$

Obtain an explicit ΔE model by substituting the state equations into the integrator equations. You obtain a model of the type:

$$\mathbf{x_{k+1}} = \mathbf{F} \cdot \mathbf{x_k} + \mathbf{g} \cdot u_k \tag{H1.2j}$$

$$y_k = \mathbf{h}' \cdot \mathbf{x_k} + i \cdot u_k \tag{H1.2k}$$

which engineers would normally call a *linear single–input, single–output (SISO) discrete–time state–space model*.

Let $h = 0.01$ *sec*, $t_f = 5$ *sec*, $u(t) = 5 \cdot sin(2t)$, $x_0 = \text{ones}(4, 1)$, where t_f denotes the final time of the simulation.

Simulate the ΔE model using MATLAB by iterating over the difference equations. Plot the output variable as a function of time.

[H1.3] Time Reversal

Given a state–space model of the form:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t) \quad ; \quad \mathbf{x}(t = t_0) = \mathbf{x_0} \quad ; \quad t \in [t_0, t_f] \tag{H1.3a}$$

which generates the trajectory behavior $\mathbf{x}(t)$.

The state–space model:

$$\dot{\mathbf{y}}(\tau) = -\mathbf{f}(\mathbf{y}(\tau), \mathbf{u}(\tau), \tau) \quad ; \quad \mathbf{y}(\tau = t_f) = \mathbf{x_f} \quad ; \quad \tau \in [t_f, t_0] \tag{H1.3b}$$

generates the trajectory behavior $\mathbf{y}(\tau)$.

Show that:

$$\mathbf{y}(\tau) = \mathbf{x}(t_0 + t_f - t) \tag{H1.3c}$$

In other words, any state–space model can be simulated backward through time by simply placing a minus sign in front of every state equation.

[H1.4] Van–der–Pol Oscillator and Time Reversal

Given the following nonlinear system:

$$\ddot{x} - \mu(1 - x^2)\dot{x} + x = 0 \tag{H1.4a}$$

This system exhibits an oscillatory behavior. It is commonly referred to as the *Van–der–Pol oscillator*. We wish to simulate this system with $\mu = 2.0$ and $x_0 = \dot{x}_0 = 0.1$.

Draw a block diagram of this system. The output variable is x. The system is autonomous, i.e., it doesn't have an input variable.

Derive a state–space description of this system. To this end, choose the outputs of the two integrators as your two state variables.

Simulate the system across 2 *sec* of simulated time. Since the system is nonlinear, you cannot use MATLAB's *lsim* function. Use function *ode45* instead.

At time $t = 2.0$ *sec*, apply the time reversal algorithm, and simulate the system further across another 2 *sec* of simulated time. This is best accomplished by adjusting the model such that it contains a factor c in front of each state equation. $c = +1$ during the first 2 *sec* of simulated time, and $c = -1$ thereafter. You can interpret c as an input variable to the model. Make sure that $t = 2.0$ *sec* defines an output point.

As you simulate the system backward through time for the same time period that you previously used to simulate the system forward through time, the final values of your two state variables ought to be identical to the initial values except for numerical inaccuracies of the simulation. Verify that this is indeed the case. How large is the accumulated error of the final values? The accumulated simulation error is defined as the norm of the difference between final and initial values.

Plot $x(t)$ and $\dot{x}(t)$ on the same graph.

Repeat the previous experiment, this time simulating the system forward during 20 *sec* of simulated time, then backward through another 20 *sec* of simulated time. What do you conclude?

1.9 Projects

[P1.1] Definitions

Get a number of simulation and/or system theory textbooks from your library and compile a list of definitions of "What is a System"? Write a term paper in which these definitions are critically reviewed and classified. (Such a compilation has actually been published once.)