Systems Modeling and Computer Simulation

Second Edition

edited by
Naim A. Kheir
Oakland University
Rochester, Michigan

Marcel Dekker, Inc. New York • Basel • Hong Kong

Library of Congress Cataloging-in-Publication Data

Systems modeling and computer simulation / edited by Naim A. Kheir.
2nd ed.
p. cm. — (Electrical engineering and electronics ; 94)
Includes bibliographical references (p.).
ISBN 0-8247-9421-4 (acid free paper)
1. Computer simulation. 2. System design. I. Kheir, Naim A.,
II. Series.
QA76.9.C65S975 1995
003—dc20 95-24572
CIP

Marcel Dekker, Inc.
270 Madison Avenue, New York, New York 10016

Current printing (last digit)
10 9 8 7 6 5 4 3 2 1

PRINTED IN THE UNITED STATES OF AMERICA

10

Computer-Aided Control System Design: Techniques and Tools

François E. Cellier The University of Arizona, Tucson, Arizona

C. Magnus Rimvall ABB Corporate Research, Heidelberg, Germany

10.1 INTRODUCTION

To this point, this book has mainly discussed diverse types of simulation techniques, and indeed, simulation has become extremely important in almost every aspect of scientific and engineering endeavor. According to Korn and Wait (1978), *simulation is experimentation with models*. Thus, each simulation program consists of two parts:

1. A coded description of the model, which we call the model representation inside the simulation program (notice the difference compared with Chaps. 1 and 2, where the term "model representation" was used to denote graphic descriptions, such as block diagrams or signal flow graphs)
2. A coded description of the experiment to be performed on the model, which we call the experiment representation inside the simulation program.

Analyzing the different types of simulation examples presented so far, it can be realized that most of these examples, independently of whether they were discrete or continuous in nature, consisted of a fairly elaborate model on which a rather simple experiment was performed. The standard simulation experiment is as follows: Starting with a complete and consistent set of initial conditions, the change in the various variables of the model (state variables) over time is recorded. This experiment is often referred to as determining the *trajectory behavior* of a model. Indeed, when the term "simulation," as is often done, is used to denote a solution technique rather than the ensemble of all modeling-related activities (as is done in this book), simulation can simply be equated to the determination of trajectory behavior. Most currently available simulation programs offer little besides efficient means to compute trajectory behavior.

Unfortunately, few practical problems present themselves as pure simulation problems. For example, it often happens that the set of starting values is not specified at one point in time. Such problems are commonly referred to as boundary value problems as opposed to the initial value problems discussed previously. Boundary value problems are not naturally simulation problems in a puristic sense (although they can be converted to

initial value problems by a technique called invariant embedding) (Tsao, 1986). A more commonly used technique for this type of problem, however, is the so-called shooting technique, which works as follows:

1. Assume a set of initial values.
2. Perform a simulation.
3. Compute a performance index, for example as a weighted sum of the squares of the differences between the expected boundary values and the computed boundary values.
4. If the value of the performance index is sufficiently small, terminate the experiment; otherwise, interpret the unknown initial conditions as parameters, and solve a nonlinear programming problem, looping through 2 . . . 4 while modifying the parameter vector to minimize the performance index.

As can be seen, this "experiment" contains a multitude of individual simulation runs.

To elaborate on yet another example, assume that an electrical network is to be simulated. The electrical components of the network are associated with tolerance values. Determine how the behavior of the network changes as a function of these component tolerances. An algorithm for this problem could be the following:

1. Consider those components with associated tolerances to be the parameters of the model. Set all parameters to their minimal values.
2. Perform a simulation run.
3. Repeat step 2 by allowing all parameters to change between their minimal and maximal values until all "worst case" combinations are exhausted. Store the results from all these simulations in a database.
4. Extract the data from the database, and compute an envelope of all possible trajectory behaviors for the purpose of a graphic display.

As in the previous example, the experiment to be performed consists of many different individual simulation runs. In this case, there are exactly 2^n runs to be performed, where n denotes the number of parameters.

These examples show that simulation does not live in an isolated world. A scientific or engineering experiment may involve many different simulation attempts. Unfortunately, the need for enhanced experimentation capabilities is not properly reflected by today's simulation software. Although model representation techniques have become constantly more powerful over the past years, little has been done with respect to enhancing the capabilities of simulation experiment descriptions (Cellier, 1986a). Some simulation languages, such as ACSL (Mitchell and Gauthier, 1991), offer facilities for model linearization and steady-state finding. Other simulation languages, such as DSL/VS (IBM, 1984), offer limited facilities for frequency domain analysis, such as a means to compute the Fourier spectrum of a simulation trajectory.

A general-purpose nonlinear programming package for curve fitting, steady-state finding, and the solution to boundary value problems, for example, is meanwhile being offered in SimuSolve (Dow Chemical, 1990), and has recently been added to the ACSL simulation environment. However, this is only one of the experiment enhancement tools that are available. For more detail, see Cellier, 1993. One should note that the current experiment enhancement tools are often difficult to use, and are very specialized and therefore limited in applicability.

Whenever such a situation is faced, we, as software engineers, realize that something may be wrong with the data structures offered in the language. Indeed, all refinements in model representation capabilities, such as techniques for proper discontinuity handling and facilities for submodel declarations, led to enhanced programming structures, whereas the available data structures are still much the same as they were in 1960s, when the CSSL specifications (Augustin et al., 1967) were first formalized.

When we talk about computer-aided design software, as opposed to simulation software, it is exactly this enhanced experiment description capability that we have in mind. Simulation is no longer the central part of the investigation, simply one software module (tool) among many others that can be called at will from within the "experiment description software," which from now on is called "computer-aided design software." Algorithms for particular purposes are called computer-aided design techniques, and the programs implementing these algorithms are called computer-aided design tools. Because many of the design tools are application dependent, our discussion is restricted to one particular application, namely, the design of control systems.

Until not too long ago, the data structures available in computer-aided control system design (CACSD) software were as poor as those offered in simulation software. However, even the available programming structures in these software tools were not helpful. Users were led through an inflexible question-and-answer protocol. Once an incorrect specification was entered by mistake, there was little chance to recover from this error. The protocol deviated from the designed path and probably led sooner or later to a complete software crash, after which the user had lost all his previously entered data and had to start from scratch.

A true breakthrough was achieved with the development of MATLAB®, a matrix manipulation tool (Moler, 1980). Its only data structure is a double-precision complex matrix. MATLAB offers a consistent and natural set of operators to manipulate these matrices. In MATLAB, a matrix is coded as follows:

```
A = [1, 2, 3; 4, 5, 6; 7, 8, 9]                    (10.1)
```

or alternatively,

```
A = [1  2  3
     4  5  6                                       (10.2)
     7  8  9]
```

Elements in different columns are separated by either a comma or a space, whereas elements in different rows are separated by either a semicolon or a carriage return. With matrices the only available data structure, scalars are obviously included as a special case. Each element of a matrix can itself again be a matrix. It is therefore perfectly legitimate to write

```
A = [0*ones(3,1),eye(3);[-2 -3 -4 -5]]             (10.3)
```

where ones(3,1) is a matrix with three rows and one column full of 1 elements; 0*ones(3,1) is thus a matrix of same size consisting of 0 elements only. Eye(3) represents a 3×3 unity matrix concatenated to the 0 matrix from the right, thus making the total

---

MATLAB® is a registered trademark of The MathWorks, Inc.

structure now a matrix with three rows and four columns. Concatenated from below is the matrix [−2 −3 −4 −5], which has one row and four columns. Thus, the preceding expression creates the matrix

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -2 & -3 & -4 & -5 \end{bmatrix} \tag{10.4}$$

Suppose it is desired to solve the linear system

$$\mathbf{A}^*\mathbf{x} = \mathbf{b} \tag{10.5}$$

For a nonsingular matrix **A**, it is known that the solution can be obtained as

$$\mathbf{x} = \mathbf{A}^{-1}*\mathbf{b} \tag{10.6}$$

which in MATLAB can be expressed as

```
x = inv(A)*b
```

or, somewhat more efficiently,

```
x = A\b
```

(10.7)

(**b** from left divided by **A**), in which case a Gaussian elimination is performed in place of the computation of the complete inverse. With MATLAB, we finally acquired a tool that allows us to learn what we always wanted to know about linear algebra [e.g., what are the eig(A+2*eye(A)) where eye(A) stands for a unity matrix with the same dimensions as **A**, and eig (. . .) computes the eigenvalues of the enclosed expression?]. In fact, such a tool had already existed for some time. It was called APL and offered much the same features as MATLAB. However, APL was characterized by a very cryptic syntax. The APL user was forced to learn to think in a fashion similar to the computer that executed the APL program, which is probably why APL never really made it into the world of applications. Moler, on the other hand, taught the computer to "think" like the human operator.

The original version of MATLAB was not designed to solve CACSD problems. MATLAB is simply an interactive language for matrix algebra. Nevertheless, this is exactly the type of tool the control engineer needs for solving problems. As an example, let us solve a simple LQG (Linear Quadratic Gaussian) regulator design problem. For the linear system

$$\frac{d\mathbf{x}}{dt} = \mathbf{A}^*\mathbf{x} + \mathbf{B}^*\mathbf{u} \tag{10.8}$$

it is desired to compute a linear state feedback such that the performance index (PI) is

$$\mathrm{PI} = \int_0^\infty (\mathbf{x}'\mathbf{Q}\mathbf{x} + \mathbf{u}'\mathbf{R}\mathbf{u})\, dt \stackrel{!}{=} \min \tag{10.9}$$

where $\mathbf{u}'$ denotes the transpose of the vector **u**. This LQG problem can be solved by means of the following algorithm:

1. Check the controllability of the system. If the system is not controllable, return with an error message.
2. Compute the Hamiltonian of this system:

$$\mathbf{H} = \begin{bmatrix} \mathbf{A} & -\mathbf{B}\mathbf{R}^{-1}\mathbf{B}' \\ -\mathbf{Q} & -\mathbf{A}' \end{bmatrix} \tag{10.10}$$

3. Compute the $2n$ eigenvalues and right eigenvectors of the Hamiltonian. The eigenvalues are symmetric not only with respect to the real axis but also with respect to the imaginary axis, and because the system is controllable, no eigenvalues are located on the imaginary axis itself.
4. Consider those eigenvectors associated with the negative eigenvalues, concatenate them into a reduced modal matrix of dimension $2n \times n$, and split this matrix into equally sized upper and lower parts:

$$\mathbf{V} = \begin{bmatrix} \mathbf{V}_1 \\ \mathbf{V}_2 \end{bmatrix} \tag{10.11}$$

5. Now, the Riccati feedback can be computed as

$$\mathbf{K} = -\mathbf{R}^{-1}\mathbf{B}'\mathbf{P} \tag{10.12}$$

where

$$\mathbf{P} = \mathrm{Re}\{\mathbf{V}_2 * \mathbf{V}_1^{-1}\} \tag{10.13}$$

The following MATLAB "program" (file RICCATI.MTL)* may be used to implement this algorithm:

```
EXEC('contr.mtl')
IF ans <> n, SHOW('System not Controllable'), RETURN, END
[v,d] = EIG([a,-b*(r\b');-q,-a']);
d = DIAG(d); k=0;
FOR j=1,2*n, IF d(j)<0, k = k+1; v(:,k) = v(:,j); END
p = REAL(v(n+1:2*n,1:k)/v(1:n,1:k);
k = -r\b'*p
RETURN
```

which is a reasonably compound way of specifying this fairly complex algorithm. Yet, contrary to an equivalent APL code, we find this code acceptably readable.

After MATLAB became available, it took amazingly little time until several CACSD experts realized that this was an excellent way to express control problems. Clearly, the original MATLAB was not designed for CACSD problems, and much had to be done to make it truly convenient, but at least a basis had been created. In the sequel, several CACSD programs have evolved: CTRL-C (Systems Control Technology, 1984; Little et al., 1984), MATRIX$_x$ (Integrated Systems, Inc., 1984; Shah et al., 1985), IMPACT (Rimvall, 1983; Rimvall and Bomholt, 1985; Rimvall and Cellier, 1985), PC-MATLAB (Little, 1985), MATLAB-SC (Vanbegin and Van Dooren, 1985), CONTROL.Lab (Jamshidi et

*The file RICCATI.MTL is expressed here in terms of the syntax of the original public domain version of MATLAB (Moler, 1980), not in terms of the more convenient syntax of the currently available commercial version of MATLAB (The MathWorks, Inc., 1992).

al., 1992), PRO-MATLAB* (The MathWorks, Inc., 1992), XMath (Gupta et al., 1993), and MaTX (Koga and Furuta, 1993). All these programs are considered "spiritual children" of MATLAB.

We want to demonstrate in this chapter how simulation software designers can learn from recent experiences in CACSD program development and how CACSD program developers can learn from experiences gained in simulation software design.

It would be convenient if a MATLAB-like matrix notation could be used within simulation languages for the description of linear systems or linear subsystems. The modeling environment Dymola (Elmqvist, 1978; Cellier, 1991; Cellier and Elmqvist, 1993) offers such a feature. In Dymola, the user can specify matrices in a convenient MATLAB-like syntax. However, because of the "horizontal sorting" capability of Dymola, matrices are expanded by the Dymola preprocessor before the sorting begins. If, for example, a 2×2 matrix A is used in a model

$$\mathbf{y} = \mathbf{A}*\mathbf{x} \tag{10.14}$$

with **A** being:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \tag{10.15}$$

meaning that

$$\begin{aligned} y_1 &= 1*x_1 + 2*x_2 \\ y_2 &= 3*x_1 + 4*x_2 \end{aligned} \tag{10.16}$$

(expanded form), it may turn out that the equations are horizontally sorted by the Dymola preprocessor into the form

$$\begin{aligned} x_1 &= 1*y_1 - 2*x_2 \\ y_2 &= 3*y_1 - 2*x_2 \end{aligned} \tag{10.17}$$

It would be useful if the simulation software could apply to linear (sub)systems an integration algorithm that is more efficient than the regularly used Runge Kutta, Adams-Bashforth, or Gear algorithms; for example, a Z transform technique currently employed for linear system simulation in PRO-MATLAB. Linear (sub)systems could automatically be identified by the modeling language compiler.

On the other hand, it is useful if all knowledge about the simulation of dynamic systems or processes is integrated into the CACSD software. Because a design usually involves more than merely simulation, it is definitely beneficial if a flexible interface between a CACSD program and a simulation language is created such that powerful simulation runs can be made efficiently at arbitrary points in a more complex design study. Such interfaces have been created for all the major contenders in CACSD software. ACSL programs (Mitchell and Gauthier, 1991) can be invoked from within CTRL-C (Systems Control Technology, 1984) and PRO-MATLAB (The MathWorks, Inc., 1992).

*Meanwhile, The MathWorks renamed PC-MATLAB and PRO-MATLAB again as MATLAB. However, we prefer to use the former names in this text to avoid confounding the commercial (C-coded) version of MATLAB with the original (Fortran-coded) version, although we frequently refer to the latter as "original MATLAB," "classic MATLAB," or "early MATLAB."

Special simulation systems with graphic front ends have also been developed. System-Build (Integrated Systems, Inc., 1987) is a nonlinear simulation environment for $MATRIX_X$, MODEL-C (Systems Control Technology, 1990) is a similar system for CTRL-C, and SIMULINK (The MathWorks, Inc., 1991) is the corresponding system for PRO-MATLAB. These three simulation environments are based on block diagrams as modeling tools.

These are some of the topics this chapter addresses. Note that this chapter discusses purely digital solutions only. Other simulation techniques (such as analog and/or hybrid simulation techniques) are discussed in Chap. 4. Although we shall not refer to these techniques explicitly any further, computer-aided control system design algorithms can be implemented on hybrid computers very easily. The dynamic process (that is, the model description) is then programmed on the analog part of the computer, and the experiment description that triggers individual simulation runs is programmed on the digital part of the computer. The digital CACSD program looks similar to the purely digital solution, and the simulation program looks exactly the same as any other analog simulation program. For these reasons, a further elaboration on these concepts can be spared.

In the next section, a systematic classification of CACSD techniques is presented. Different techniques (algorithms) for computer-aided control system design are discussed.

In Sec. 10.3 CACSD tools are classified. This discussion highlights the major differences between several classes of CACSD tools.

Both Secs. 10.2 and 10.3 help to prepare for Sec. 10.4, in which a number of currently available CACSD tools are compared with respect to features (algorithms) offered by these software systems.

The problem of software standardization versus software diversification is discussed in Sec. 10.5.

In Sec. 10.6, we show how simulations can intelligently be used within CACSD software. This section helps to create a bridge to other chapters in this book.

Finally, Sec. 10.7 presents our perspective of current and forthcoming developments in the area of CACSD software design.

10.2 DEVELOPMENT AND CLASSIFICATION OF CACSD TECHNIQUES

Let us look briefly into the history of CACSD problems. CACSD, as we know it today, has its roots in a technology that was boosted by the needs created in World War II, when military leaders started to think about more powerful weaponry, and engineers produced answers in the form of automatically controlled, in place of manually controlled, weapons systems. (Fortunately, automatic control has since found many other nonmilitary applications as well. Nevertheless, even today, a substantial percentage of research grants in the automatic control field stems either directly or indirectly from national defense sources, even though funding opportunities for research in automatic control can become limited at times.)

In the beginning, that is, in the 1930s through 1950s, engineers were dealing with isolated (small) continuous-time systems with one single input and one single output (so-called SISO systems). The design of these systems was (at least here in the West) predominantly done in the frequency domain, most prominently represented by such people as Evans and Nyquist. Most of the techniques developed were graphic in nature.

With the need to deal with more complex systems with multiple inputs and outputs (so-called MIMO systems), these graphic techniques failed to provide sufficient insight. It was, among others, Kalman who led the scientists and engineers back into the time domain, where systems were now represented in the so-called state space, that is, by sets of first-order ordinary differential equations (ODEs) in place of *n*th-order ODEs. For further detail, refer to Chap. 2 of this book and to its references. This modern representation seemed to be better amenable to a systematic (algorithmic) design methodology. This representation was very naturally extensible from SISO system representations to MIMO system representations, and many of the algorithms (such as LQG design) would work as well on MIMO systems as on SISO systems. With the advent of modern digital computers, it was possible to apply these algorithms also to "higher" order systems (say, fifth- to tenth-order systems), whereas the previous hand computations were limited to second- to third-order systems. (This was actually the major reason for choosing a frequency domain representation in previous decades: frequency domain design can also be done for higher order systems.) This was the technology of the 1960s.

What has happened since? What were the major breakthroughs in the 1970s and 1980s? Earlier research in control theory was fairly well consolidated; with diversification, different types of approaches were made available to tackle different types of problems.

One of the major drawbacks of the previously used technology was, ironically, found in the high degree of automation characterizing its algorithms. One jotted down some values and called on a subroutine, and the answer came in the form of other numbers, parameter values, or gain factors, for example. The procedure was "sterile." Somehow lacking was the intuitive feel for what was going on. What if the LQG design failed to produce acceptable answers? Where did we go from there? Often, the conclusion was that the structure of the chosen controller was inappropriate for the task, and thus optimization of the parameters of the inappropriate controller was doomed to failure. Therefore, the control engineer had to take structural decisions in place of purely parametric ones. Unfortunately, such decisions can hardly be taken without profound insight into what was going on. None of the automated algorithms available at that time were able to determine an adequate controller structure.

For these reasons, several researchers went back to the frequency domain and came up with some new design tools [such as a generalization of the Nyquist diagram (Rosenbrock, 1969)], and some new system representations [such as some varieties of polynomial matrix representations (Wolovich, 1974; Wonham, 1974)]. Other groups decided on a different approach to tackle the same problem. Instead of producing individual solutions in the form of sets of parameter values, they tried to develop algorithms that would produce entire "fields" of output parameters as a function of input parameters, to come up with, for example, three-dimensional graphs in the parameter space. For instance, this is often done in the so-called robust controller design (Ackermann, 1980). Unfortunately these techniques usually involve multiple sweeping, which is number crunching at its worst. Fortunately, the sheer increase in the power of modern engineering workstations makes approaches feasible and even reasonable that would have been quite unexpected only a few years ago. An elegant and efficient modern implementation of parameter sweeping techniques is provided in Boyd and Barratt (1991).

For a somewhat less expensive solution, it may be possible to employ sensitivity analysis instead (Cellier, 1986a). Recent developments in this area try to do away with numeric algorithms altogether. Instead of computing numerically one point in the parameter space at a time, the new algorithms reproduce what the engineer once did in the

paper and pencil age, that is, formula manipulation. The latest developments in nonnumeric data processing are employed to obtain algorithmically and automatically a formula that relates the designed output parameters to the given input parameters. These techniques are still in their infancy, however, and no commercial product of this kind is available as of today. The most exciting new development in this area represents Dymola (Elmqvist, 1978; Cellier and Elmqvist, 1993; Elmqvist, 1994), a modeling language with powerful symbolic formula manipulation capabilities. More on Dymola is presented later.

Another development was initiated by the need to deal with even larger systems. How do you control a large system consisting of many subsystems in an "optimal" way? Many of the previously used algorithms fail to work properly when applied to 50th- or 200th-order systems. They either compute forever, fail to converge, or produce a result that is accurate to exactly zero significant digits! One way to tackle this problem is to try to partition the system into smaller subsystems and find answers for those subsystems first. This led to decentralized control (Athens, 1978) and hierarchic control (Siljak and Sundareshan, 1976) schemes. More information about these approaches can be found in Chap. 14 of this book. Another answer, of course, might be to design new centralized algorithms that work better on high-dimensional systems (Laub, 1980).

The availability of reliable low-cost microprocessors led to the need to implement subsystem controllers by a digital computer. This stimulated research into discrete-time algorithms: the continuous-time algorithms applied to discrete-time systems tend to exhibit very poor stability behavior.

Finally, the new age of robotic technology led to the need to develop better algorithms for the control of nonlinear systems (Asada and Slotine, 1986). The models describing the dynamics of robot movements are highly nonlinear. Most of the more refined algorithms that were previously developed work poorly, or not al all, when applied to nonlinear systems. Unfortunately, the robustness of an algorithm is often inversely proportional to its refinement; that is, the more specialized an algorithm, the less likely it is able to handle modified siutations. One way to solve this problem is to view the nonlinear time-invariant system as a linear time-variant system and to design control algorithms for this class of problems, such as selftuning regulators (Åström, 1980), model-reference adaptive controllers (Monopoli, 1974; Narendra, 1980), and robust controllers (Ackermann, 1980). Two major breakthroughs in the design of complex nonlinear systems were accomplished in the late 1980s and early 1990s. One relates to the use of neural networks for control (Narendra and Parthasarathy, 1990; Karakasoglu, 1991; Cellier and Pan, 1995); the other involves the use of fuzzy controllers (Pedrycz, 1989; Jamshidi et al., 1993; Kandel and Langholz, 1994; Cellier and Mugica, 1995).

So far, we have presented the problems to be solved. Problems can be classified into single-input/single-output, multiple-input/multiple-output, and decentralized problems. For each class of problems, a different suite of algorithms was developed to solve them. Until now, we have totally ignored the problem of the numeric aptness of an algorithm, of numeric accuracy and numeric stability. The numeric behavior of algorithms is highly dependent on the system order, that is, the number of state variables describing the system or process. Almost any algorithm can be used to solve a 3rd-order problem. Many algorithms fail when applied to a 10th-order problem, and almost all of them fail to solve a 50th-order problem correctly. This is true for almost every algorithm in all three classes of problem types. Since the late 1970s, many researchers, including Moler, Golub, Laub, Wilkinson, and van Dooren, have designed a series of new algorithms for SISO and

MIMO system design that are less sensitive to the system order. A major breakthrough in this area was the development of the singular value decomposition described in Golub and Wilkinson (1976).

From now on, algorithms that work only for low-order systems are referred to as LO algorithms, techniques that also work for high-order systems are called HO algorithms, and finally, methods that can be used to treat very high order systems (mostly discretized distributed parameter systems) are called VHO algorithms.

Let us introduce next the concepts used in the design of the different classes of algorithms more explicitly. Most of the algorithmic research done so far was concerned with algorithms based on canonic forms (Kailath, 1980). All these canonic forms, in turn, are based on minimum parameter data representations. What is a minimum parameter data representation? A SISO system can be represented in the frequency domain through its transfer function.

$$g(s) = \frac{(b_0 + b_1 s + \cdots + b_{n-1}s^{n-1})}{(a_0 + a_1 s + \cdots + a_{n-1}s^{n-1} + s^n)} \tag{10.18}$$

where the denominator polynomial is of nth order (the system order) and the numerator polynomial is of $(n-1)$st order. This representation is unique; that is, the system has exactly $2n$ degrees of freedom (the degrees of freedom equal the number of linearly independent parameters of any unique data representation). The parameters of this representation are the coefficients of the numerator and denominator polynomials. Any set of parameter values describes one system, and no two different sets of parameters describe the same system. Any data representation sharing this property is a minimum parameter representation. The controller-canonic representation of this system can be written as

$$\dot{\mathbf{x}} = \begin{bmatrix} 0 & 1 & 0 & 0 & \ldots & 0 \\ 0 & 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 0 & 1 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ -a_0 & -a_1 & -a_2 & -a_3 & \ldots & -a_{n-1} \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \ldots \\ 1 \end{bmatrix} u \tag{10.19}$$

$$\mathbf{y} = \begin{bmatrix} b_0 & b_1 & b_2 & b_3 & \ldots & b_{n-1} \end{bmatrix} \mathbf{x}$$

Counting the number of parameters of this representation, it can easily be verified that this representation has exactly $2n$ parameters, and they are the same as before. Also, the Jordan-canonic representation

$$\dot{\mathbf{x}} = \begin{bmatrix} \lambda_1 & 0 & 0 & 0 & \ldots & 0 \\ 0 & \lambda_2 & 0 & 0 & \ldots & 0 \\ 0 & 0 & \lambda_3 & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & 0 & \ldots & \lambda_n \end{bmatrix} \mathbf{x} + \begin{bmatrix} 1 \\ 1 \\ 1 \\ \ldots \\ 1 \end{bmatrix} u \tag{10.20}$$

$$\mathbf{y} = \begin{bmatrix} r_1 & r_2 & r_3 & r_4 & \ldots & r_n \end{bmatrix} \mathbf{x}$$

(assuming all eigenvalues λ_i to be distinct) has exactly the same number of parameters. This is true for all canonic forms. For LO systems, these representations are perfectly acceptable. However, we require redundancy to optimize the numeric behavior of algorithms for HO systems. Thus, all algorithms that are based on canonic forms are clearly LO algorithms.

HO algorithms can be obtained by sacrificing this "simple" system representation through the introduction of redundancy. These new system representations contain more than $2n$ parameters with linear dependencies existing between them. This redundancy can now be used to optimize the numeric behavior of control algorithms by balancing the sensitivities of the parameters (Laub, 1980). Some of the better HO algorithms are based on Hessenberg representations (Patel and Misra, 1984).

VHO systems (that is, systems of higher than about 50th order) typically result from discretization of distributed parameter systems. Most of the algorithms developed for this class of systems until now exploit the fact that, in general, VHO systems have sparsely populated system matrices. Thus, algorithms have been designed that address matrix elements through their indices. Careful bookkeeping ensures that only elements that are different from zero are considered in the evaluations. These so-called sparse matrix techniques are associated with a certain overhead. Thus, they are not cost effective for the treatment of LO systems, and even many HO systems are handled more efficiently by the regular algorithms. As a rule of thumb, sparse matrix techniques become profitable for systems of higher than about 20th order.

Contrary to the algorithms for HO problems just described, sparse matrix techniques do not influence the numeric behavior of the involved algorithms, only their execution time. Therefore, the numeric problems discussed for HO systems remain the same. (In most of the published papers discussing VHO problems, sparse matrix techniques have been applied to one or another of the classic canonic forms.) Unfortunately, the introduction of redundancy also reduces the sparsity of the system matrices and eventually annihilates it altogether. Therefore, these two approaches are in severe competition. More research is needed to find a solution to this serious problem.

Most of the research described so far was concerned with time domain algorithms. It has often been said that frequency domain operations are numerically less stable than time domain operations and it is believed that this statement is incorrect. It is not the frequency domain per se that makes the algorithms less suitable; it is the data representation currently used in frequency domain operations that has these undesirable effects. As previously discussed, if one wants to minimize the numeric sensitivity of an algorithm, one must balance the sensitivities of the system parameters; that is, each output parameter should be about equally sensitive to changes in the input parameters (Laub, 1980). Also, the sensitivities of algorithmic parameters should be balanced. In the time domain, this has been achieved by the process of orthonormalization, by operating on Hermitian forms (Golub and Wilkinson, 1976). In the frequency domain, it is less evident how the balancing of sensitivites can be achieved. If we represent a polynomial through its coefficients, even the evaluation of the polynomial at any value of the independent variable with a norm much larger or much smaller than 1 leads to extremely unbalanced parameter sensitivities. Let us consider the polynomial

$$q(s) = a_n s^n + a_{n-1} s^{n-1} + \cdots + a_1 s + a_0 \tag{10.21}$$

If this polynomial is evaluated at $s = 0$, obviously the only parameter that has any influence on the outcome is a_0; that is, a_0 is sensitive to this operation, but all other parameters are not. However, if we evaluate the polynomial at $s = 1000$, obviously a_n exerts the strongest influence, and a_0 can easily be neglected. This problem disappears when we represent the polynomial through its roots:

$$q(s) = k(s - s_1)(s - s_2) \cdots (s - s_n) \tag{10.22}$$

Here, the parameters are (k) and $(s_1 \ldots s_n)$ instead of $(a_0 \ldots a_n)$. However, if we want to add two polynomials, we do not get around to (at least partially) defactorizing the polynomials and refactorizing them again after performing the addition. The processes of defactorization and refactorization have badly balanced sensitivities and are thus numerically harmful.

Traditionally, these were the only two data representations considered, and both are obviously unsatisfactory. There is little we can do to improve the numeric algorithms based on these data representations: both are minimum parameter representations. However, we have other choices. For instance, it is possible to represent a polynomial through a set of supporting values. Let us evaluate $q(s)$ at any $n + 1$ points. If we store these $n + 1$ values of s together with those of $q(s)$, we know that there exists exactly one polynomial of nth order that fits these $n + 1$ points. We can "reconstruct" the polynomial at any time (that is, find its coefficients), and this therefore gives rise to another data representation (Sawyer, 1986). If we choose more points, we can make use of redundancy and reconstruct the polynomial by regression analysis, reducing the numeric errors involved in this computation. The basic operations (addition, subtraction, multiplication, and division) all become trivial in this data representation if all involved polynomials are evaluated over the same base of supporting values (we merely apply them to each data point separately), and because most algorithms are based solely on repeated application of these basic operations, they can also be performed easily within this data representation. The redundancy inherent in this data representation can eventually be used to balance parameter sensitivities by selecting the supporting values (values of s) carefully. A preliminary study (Sawyer, 1986) indicates that the best choice might be to place the supporting values equally spaced along the unit circle. More research is still needed, but it is believed this approach might lead to a breakthrough in the numeric algorithms for frequency domain operations. Another data representation that shares most of the numeric properties of supporting value representation is to interpret the coefficients of the polynomial as a vector, compute the fast-Fourier transform of that vector, and use the transformed vector as the base representation for polynomial and polynomial matrix operations (Chi and Cellier, 1991).

To summarize this discussion, CACSD techniques can be classified in several ways: techniques for SISO, MIMO, and decentralized systems; techniques for frequency versus time domain operations; techniques for continuous-time versus discrete-time systems; techniques for linear versus nonlinear systems; and, finally, techniques for low-order, high-order, and very high order systems.

Another classification distinguishes between user-friendly and non–user-friendly algorithms; user-friendly algorithms allow us to concentrate on physical design parameters rather than on algorithmic design parameters. As a typical example of user-friendly algorithms, we may mention the variable-order, variable-step integration algorithms, which enable us to specify the required accuracy (a physical design parameter) as opposed to the integration step length and order (which are algorithmic design parameters).

Finally, one should distinguish between numeric and nonnumeric algorithms. Nonnumeric algorithms make use of formula manipulations and "reasoning" techniques that are usually connoted as artificial intelligence (AI) techniques. Because none of the CACSD programs discussed in Sec. 10.4 makes extensive use of such techniques, we shall save a more intimate discussion of AI techniques for Sec. 10.7 on outlook.

10.3 DEVELOPMENT AND CLASSIFICATION OF CACSD TOOLS

Although control theory as we know it today is a child of the early part of this century, computer-aided control system design really did not start until 1970. At that time, it would take roughly half a day merely to find the eigenvalues of a matrix because this involved the following procedure:

1. Develop a program to calculate eigenvalues by calling a library subroutine with about six call parameters ($^1/_2$ h).
2. Walk to the computer center to prepare the data input (20 minutes).
3. Wait for a card puncher to become available ($^1/_2$ h).
4. Prepare input data (10 minutes).
5. Submit card deck to input queue, and wait for output to be returned (turnaround time roughly 1 h).
6. Correct typographic errors after waiting for another card puncher to become available, and resubmit card deck; wait again for output (90 minutes).
7. Walk back to office (20 minutes).

The solution of a true control problem (e.g., the simple LQG design problem described earlier) took a considerably longer time, possibly as much as 1 or 2 weeks. No wonder most colleagues detested the computer at that time and preferred to specialize in other topics that did not require involvement in this denervating process.

Around 1972, the writers undertook the effort to ask colleagues in the department not to throw away their control programs (after they were done with a particular study), but rather document their subroutines and hand them over for inclusion in a "control library" to be built. By 1976, an impressive (and somewhat formidable) set of (partially debugged) control algorithms had been collected (Cellier et al., 1977). At this time, we decided to ask colleagues from other universities to join in the effort and share their control routines and libraries with us as well. We started the PIC service, a program information center for programs in the control area, and circulated a short newsletter twice a year providing information in the form of a "who has what" in control algorithms and codes. Meanwhile, as first computer terminals became available to us, and using our control library, we were able to reduce the time needed to solve most (simple) control problems to 1 or 2 days of work.

At that time, it was considered important to work toward reducing further the turnaround time by creating an interactive "interface" to our control library. This required conversion of the program to a PDP 11, because the CDC machine of the computer center could be used for batch operation only. Clearly, the interface was meant to be a relatively small add-on to our library, and most of our effort and time were spent in improving the control subroutines themselves. Nevertheless, this activity resulted in INTOPS (Agathoklis et al., 1979), one of several interactive control system design programs made available around the same time. The first generation of true CACSD programs was, however, very limited in scope. To keep the interface simple, the programs were strictly question-and-answer driven, with the effect that they were almost useless for research. True research problems simply do not present themselves in the form of classroom examples that follow a prepaved route as foreseen by the developers of the CACSD software. INTOPS proved very useful for undergraduate control education, though. Suddenly, the use of computers became real fun to many. As a research tool,

however, INTOPS failed to provide the necessary flexibility, and it became clear that a true full-fledged programming language was required for this purpose. Unfortunately, such a language could no longer be considered a small and inexpensive add-on to the control library.

In the fall of 1980, Åström and Golub undertook the commendable effort to bring recognized numeric analysts and control experts together in the first conference on numeric techniques in control ever held. On this occasion, we met with Moler, who demonstrated his newly released MATLAB software. It took us only minutes to realize the true value of this instrument for our task. When we returned to Zurich, we implemented MATLAB first on a PDP 11/60, and a short while later on the freshly acquired VAX 11/780. Within 1 year, MATLAB became the single most often used program on that machine (which belonged to the department of electrical engineering). Students were able to learn the use of this tool within half an hour, and suddenly, researchers also became interested in our "gadgets." MATLAB was fully command driven.

An often-heard criticism of command-driven languages is that they are too complicated for the occasional user. Who can remember all those commands and their parameters except someone who uses the tool on a daily basis? This was simply not true. Our students were enchanted, and they found MATLAB actually much easier to use than the question-and-answer–driven INTOPS program. Extensive interactive HELP information is available to aid in the use of any particular function, and this proved completely satisfactory to our users.

As noted earlier, the original MATLAB was not designed to be a CACSD tool. There are many shortcomings of the early MATLAB for our purpose. These were summarized as follows (Cellier and Rimvall, 1983):

1. The programming facility (EXEC-file) of MATLAB is insufficient for more demanding tasks; EXEC-files have no formal arguments; EXEC-files cannot be called as functions but only as subroutines; WHILE, FOR, and IF blocks cannot be properly nested; there is neither a GOTO statement nor an (alternative) loop exit statement; and the input/output capabilities of EXEC-files are too limited.
2. The SAVE/LOAD concept of MATLAB is insufficient: this immediately results in large numbers of files that are difficult to maintain in an organized fashion. A true database interface would be valuable. Moreover, users want the possibility to interface data produced or used by their own programs with MATLAB.
3. Control engineers prefer results in graphic form. The output facilities offered by MATLAB are insufficient in every respect. A database interface would at least soften this request: it would allow the use of a separate stand-alone program, outside MATLAB, to view data produced by MATLAB graphically.
4. MATLAB does not lend itself easily to operations in the frequency domain.
5. Many control systems call for nonlinear controllers (e.g., windup techniques for treatment of saturations and adaptive controllers for time-varying systems). MATLAB does not provide a mechanism to describe nonlinear systems.
6. A library of good and robust control algorithms is needed. (This final request is actually the one that is easiest to satisfy.)

It should be noted that these shortcomings relate to the original (classic) version of MATLAB, and not to the more recent commercial version, where most of these shortcomings have been addressed and removed.

In the sequel, a number of CACSD programs were made available that provide answers to one or several of these demands. These (and others) are reviewed in the following section.

In summary CACSD tools can be classified into sub-program libraries versus integrated design suites. The first generation of CACSD tools was of the former type; the more recent are mostly of the latter type. This new type of CACSD tool can be further classified as either comprehensive design tools or design shells. The former type tries to provide algorithms that handle all imaginable control situations. This may eventually result in very large programs offering many different features; KEDDC (Schmid, 1979, 1985) is an example of this type. The design shells type provides an open-ended operator set that allows the user to code his or her own algorithms within the frame of the CACSD software; MATLAB (Moler, 1980) is an example of this type. Of course, a combination of these two categories is possible (and probably most useful) to the control engineer.

CACSD programs can furthermore be either batch operated or fully interactive, or both. The interactive mode is useful for a quick analysis and understanding of what is going on in a particular project. However, there are many control design problems, such as optimal design of nonlinear systems, that call for an extensive amount of number crunching. These problems are best executed in batch mode.

CACSD program can be either code driven or data driven or a combination of the two (e.g., by incremental compiler techniques). Code-driven programs are compiled programs that implement their algorithms and operators in program code. They are faster executing, but they are less flexible and less easy to augment. On the other hand, data-driven programs implement algorithms and operators as data statements interpreted during program execution. They are powerful tools for experimentation, but not necessarily for production. It is usually a good idea to develop a new CACSD software first as a data-driven program. Later, once the features and format of the new software are stabilized, it can be reimplemented as a code-driven program for improved efficiency. Compilers can eventually be used to (at least partially) automate the step from the data-driven to the code-driven implementation.

The user interface of CACSD programs can be question and answer driven, command driven, menu driven, form driven, graphics driven, or window driven. In a question-and-answer-driven program, the user is asked questions to determine what must be computed next. Thus, the program flow is completely pre-determined. This type of user interface is easiest to implement, but it is inflexible and probably not very useful in a research environment.

Newer CACSD programs are often command driven. Here, the initiative stays completely with the user. The CACSD program sends a "prompt" to the terminal indicating its readiness to receive the next command in the same manner as an interactive operating system (e.g., VMS or Unix) would. In fact, an interactive operating system is nothing but a command-driven interactive program. Turning the argument over, command-driven CACSD programs can also be viewed as special-purpose operating systems. One disadvantage of this type of user interface is the need to remember what commands are available at every interface level. This problem is today mostly remedied by providing an extensive interactive help facility.

Another alternative is to use a menu-driven interface. Here, the CACSD program displays a menu of the currently available commands on the screen instead of merely sending a prompt. It then waits for the user to choose one of the items on the list, normally by use of a pointing device, such as a crosshair cursor or a mouse. This interface type is quite easy to implement, and it can be very powerful. One of its major

drawbacks is the amount of information that must be exchanged between the program and the user.

A form-driven interface is most profitably used during the setup period of the CADSD program for supplying (or modifying) default values for large numbers of defaulted parameters of more intricate CACSD commands or operators. Here, the screen is split into separate alphanumeric fields. Each field is used to supply one parameter value. The user can jump from one field to the next to supply (override) parameter values. This interface requires a direct addressing mode to position the alphanumeric curser on the screen. Although there meanwhile exists an American National Standards Institute (ANSI) standard for this task, many hardware manufacturers already offered such a feature when the standard became available and refused to modify their hardware and system software to comply with the standard. A laudable exception is Digital, which adopted the ANSI standard when switching from the VT52 series terminals to the VT100 series terminals. Most modern graphics workstations support XTerm, a protocol that is VT100 compatible.

A graphics-driven interface was originally used to display results from a CACSD analysis, such as a Bode diagram or a simulation trajectory, in a graphic form (and this is about all that can be done on a mainframe computer with a serial asynchronous user interface). However, one obstacle has always been the high degree of terminal dependence of any graphics solution. One way to overcome this problem was to employ a graphics library providing for a large variety of terminal drivers to be placed between the CACSD software and the terminal hardware. In the past, several such libraries were developed (e.g., DISSPLA and DI-3000). Unfortunately, all these commercially available libraries were expensive, and no true standard existed. Advanced graphics call for high-speed communication links. Meanwhile, special-purpose graphics workstations have been developed (e.g., APOLLO Domain and Sun) that provide the necessary speed for enhanced graphics capabilities. Such workstations support X, a recent low-level window graphics protocol that has become an industry standard. On top of X, a higher level window graphics protocol by the name of Motif has also become a de facto standard and is supported on most engineering workstations. However, even simple personal computers (PCs) have meanwhile become so powerful that they can be used almost interchangeably with workstations. Contrary to the engineering workstations, PCs support another window graphics standard called Microsoft Windows. Even higher level software, such as C++/Views, has recently become available that allows programming graphic applications at an even higher abstraction level. C++/Views can then be compiled down to either Motif or Microsoft Windows, making C++/Views applications even more hardware independent.

A first generation of simulation programs was meanwhile made available that offers a true animation feature. A mechanism is provided to synchronize the simulation clock with real time, and the user can watch results from a simulation either on-line (that is, while the simulation is going on) or off-line (that is, driven from the simulation database), in a true relation to real time (slower than, equal to, or even faster than real time). For an increased feeling of reality, the color graphics screen is divided into a static background picture and an overlaid dynamic foreground picture on which the simulation results are displayed. TESS (Standridge and Pritsker, 1987) and CINEMA (Systems Modeling Corp., 1985) are two such programs. Some flight simulators use a "wallpaper" concept to make the background picture even more realistic. Polygons can now be filled with patterns that represent a blue sky with slight haziness and a few fluffy clouds or a green meadow with flowers and some trees. Here, the background picture is partly dynamic as

well, fed from a three-dimensional database, and a projection program automatically calculates the currently visible display (Evans, 1985). This research led more recently to amazingly powerful virtual reality environments. Virtual reality is only another word for high-level real-time simulation with hardware (a human) in the loop and with extremely powerful three-dimensional animation facilities. For trends in visualization, see Sect. 5.5.

Graphic input has also become a reality. Control circuits can be drawn on the screen as block diagrams, which are then automatically translated by a graphic compiler into a coded model representation. MATRIX$_X$ (Integrated Systems, Inc., 1984, Shah et al., 1985) already provides this feature when operated from a Sun workstation (the program module implementing this feature is called SystemBuild). The most fancy implementation, however, is provided in HIBLIZ (Elmqvist, 1982; Elmqvist and Mattsson, 1986). This program uses a virtual screen concept similar to that used in a modern spreadsheet program. The virtual screen is a portion of memory that maintains the entire graph. The physical window can be "moved" over the virtual screen such that only part of the total graph is visible at any one time. A zoom feature is provided to determine the percentage of the virtual screen to be depicted on the physical screen. The program is hierarchic. Breakpoints are used to determine the amount of detail to be displayed. In a typical application, when the entire virtual screen is made visible, a box may be seen containing a verbal description of the overall model. Once the user starts to zoom in on the model, a breakpoint is passed at which the previously visible text suddenly disappears and is replaced by a diagram showing a couple of smaller boxes with interconnections between them. When the user zooms in on one of these (so far empty) boxes, a new text may suddenly appear that describes the model contained in this box, and so on. At the innermost level, boxes are described through sets of differential equations, a table, a graph, a transfer function, or a linear system description. The connections (paths between boxes) are labeled with their corresponding variable names placed in a small box located at both ends of the path. Pointing to any of these variables, all connections containing this variable are highlighted. During compilation, an arrow points to the part of the graph that is currently being compiled. During simulation, the user can zoom in on any path until the "small box" containing the names of variables in this path becomes visible. Pointing to any of these variables now, the user immediately obtains a display of a graph of that variable over time.

One should note that at the time when HIBLIZ was designed, neither the then available computer hardware nor the necessary implementation software were adequate for the task at hand; thus, HIBLIZ is no longer in use. However, a recent replacement with similar features, based on high-resolution screens, professionally-made window software, and object-oriented programming, is Dymodraw (Dynasim, 1995).

A window-driven interface permits splitting the physical screen into several logical windows. Each window is now associated with one logical unit in the same manner as different physical devices once were. Each window by itself can theoretically be alphanumeric or graphic, question and answer driven, command driven, menu driven, or form driven. Thus, the window interface is actually at a slightly different level of abstraction than the previously described interface types. Windows can often be overlaid. In this case, the most recently addressed window automatically becomes the top window, which is completely visible. On some occasions, windows are attached to a logical screen. The concept here is to allow multiple screens that can be pulled down on the physical screen in a fashion similar to a roll shutter. In practice, window management calls for high-resolution bit-mapped displays (at least 700 $\times$ 1000 pixels).

The different interface modes just described are by no means incompatible. In IMPACT (Rimvall, 1983; Rimvall and Bomholt, 1985; Rimvall and Cellier, 1985), we experimented with combinations of several interface types. IMPACT is largely command driven. However, in IMPACT, an extensive query facility is available that goes far beyond the interactive help facility offered in previous programs. By use of the query feature, the user can obtain guidance at either the individual command level or the entire session level; thus, one can decide on an almost continuous scale at which level of guidance to operate the software (with the pure question-and-answer-driven mode as the one extreme and the pure command-driven mode as the other). A form-driven interface is being provided for particular occasions, for example, to determine the format of graphs to be produced by IMPACT. A window-driven interface is provided for the management of multiple sessions. Multiple sessions are created by a SPAWN facility that works similarly to that provided in VAX/VMS. However, our SPAWN facility goes far beyond that of VMS. At any instant, even while entering parameters to a function, the user can SPAWN a new subprocess as a scratchpad for intermediate computations.

These interface discussions do not pertain to CACSD programs alone. In fact, they are crucial considerations in any interactive program. The most modern operating systems experiment with precisely the same elements. For instance, the operating system of the MacIntosh can be classified as a window-driven graphic operating system in which the windows themselves are sometimes menu driven and sometimes form driven.

10.4 CACSD TOOLS—A SURVEY

In the first edition of this book, a bewildering multitude of CACSD programs were discussed. This was justified then, since indeed, no true de facto standard had been established yet, although even at that time, it was suspected that most of these programs would fade before long. This suspicion has become true. Most of the previously mentioned programs are no longer in use. Thus we decided to scrap this entire section of the chapter, and replace it by what can be conceived as the current state-of-the-art in CACSD software.

MATLAB and its brethren indeed made it! The MATLAB syntax (inherited from the original MATLAB by all of its versions) is so convenient and natural to use that there can hardly be any reason why someone would not wish to employ it.

However, even among the many children of the original MATLAB, there are only two true survivors: modern MATLAB (formerly PRO-MATLAB and PC-MATLAB) and Xmath (formerly MATRIX_X). The market for such products has (to the benefit of their end users) become so competitive that only products with a strong commercial outfit behind them are able to continue in use.

Both companies realized several years ago that the simulation market is vitally important. Whereas controllers for linear systems can be designed within the framework of linear algebra alone (the trademark of the early years), controllers for nonlinear systems must be designed in a trial-and-error fashion, by iterating over simulation runs. Thus, it is essential that CACSD software be able to call upon a simulation engine. To this end, MATRIX_X was equipped with a simulation engine called SystemBuild, and MATLAB was enhanced by SIMULINK.

Just as their parent programs, SystemBuild and SIMULINK are like twin brothers. Each of the two software systems consists of three separate modules. In both systems, the user models a nonlinear continuous-time plant with continuous or discrete-time controllers using a graphical block-diagram description. The block diagram is then compiled into an internal representation by a graphical compiler. The internal representation is finally simulated by a code interpreter. Both systems offer a fairly slick and intuitive user surface. However, block diagrams are not truly modular as we shall discuss in due course, and the technology used is less geared toward large industrial systems. Also, the run-time execution is comparatively slow because the model is usually interpreted rather than compiled. Again, for small systems this is not a serious problem, but large systems are much better simulated in more advanced simulation software such as ACSL.

Both companies have realized this shortcoming, and offer options for true compilation of simulation programs. For example, The MathWorks offers a program called the "SIMULINK Accelerator" that compiles SIMULINK graphics into C code to be linked with MATLAB. However, the facility is still of limited power in comparison with e.g. ACSL, since user-written code (so-called S-functions) are not compiled, and hand-encoding models directly in C instead of using S-functions, while manually retrieving the input variables from the MATLAB-maintained stack and depositing the output variables back onto the stack by referencing required MATLAB functions in their C-format, is rather painful.* In addition, it is paramount that a continuous-system simulation language offer event handling capabilities to describe discontinuous models. ACSL offers such a facility that works satisfactorily well. The newest versions of SIMULINK and SystemBuild have added some limited event handling capabilities also. In SIMULINK, for example, the step-size control of the integration algorithm is used to locate state events, rather than offering proper event indicator functions. This is both dangerous and inefficient (Cellier, 1995). Another drawback is that SIMULINK doesn't offer means to express discrete actions to take place in response to events.

Yet overall, both MATLAB and XMath are excellent products, and, because of their fierce competition, are marketed at reasonable prices. One of these software tools should be in the "software toolbox" of every engineer today. Of the two products, MATLAB has more users. In 1986, it was just the other way around. The reason is that The MathWorks, wisely, made stronger efforts at opening up the architecture. It is an essential feature of MATLAB that external users can generate MATLAB toolboxes that are indistinguishable in appearance from system-maintained function libraries. In particular, the documentation of the functions contained in these toolboxes is immediately and automatically accessible through the MATLAB help facility.

Indeed, a number of second-source software developers have already created a wealth of powerful MATLAB toolboxes for various aspects of control system design as well as other purposes. Most of these are marketed through The MathWorks, who offer to market MATLAB toolboxes for any second-source software developer, again a wise decision.

A series of textbooks for control education offer either an entire chapter on CACSD software or at least an appendix, and a number of textbooks have recently appeared that are basically CACSD software manuals (see the references at the end of this chapter). Almost without exception, these books advertize the use of MATLAB (Hanselman and Kuo, 1995; Ogata, 1994a, 1994b). In this way, MATLAB has found its way into the

*This problem will be remedied by the new M-file to C-code compiler.

university classrooms, and into the brains of the next generation of practicing control engineers.

It is believed that the future in CACSD software is in the direction of open architectures that standardize the software interfaces, rather than the tools themselves (Barker, 1994), and MATLAB's source of success is that The MathWorks bought early on into this concept. Educators choose MATLAB over other products because they are enabled and encouraged to develop their own MATLAB toolboxes, which they can then present together with all the other toolboxes that are already on the market. Open architectures are clearly the way of the future.

However, Xmath is by no means out of acceptance or use. A serious drawback of MATLAB is that it still offers no means to declare variables. Xmath has superior capabilities in this respect. It is somewhat inconvenient for a designer of toolboxes to have to rely on matrices alone to represent data structures, as in the course of designing algorithms for MIMO systems in the frequency domain. Also SystemBuild has some important advantages over SIMULINK. In particular, it already supports differential algebraic equations.

Is there nothing else out there worth mentioning? Evidently, neither MATLAB nor Xmath have much to offer in terms of symbolic processing.* Whereas some general-purpose symbolic formula manipulation tools, such as Mathematica, are widely used, they are not suited for use in large-scale control system design. Even on a fast processor, the design is time-consuming, and, yet more devastatingly, the formulae generated by Mathematica (and similar tools) have a tendency to explode. What is needed are much more specialized tools for symbolic model manipulation.

Such tools have meanwhile become available. Dymola (Cellier and Elmqvist, 1993; Elmqvist et al., 1993) is perhaps the most exciting new development on the modeling, simulation, CACSD, and concurrent engineering software markets. Originally a university prototype (Elmqvist, 1978), Dymola recently grew into a fully-supported industrial product (Elmqvist, 1994).†

Dymola was built on the premise that physics doesn't understand the difference between cause and effect as long as they are simultaneous. There is no physical experiment that allows us to determine whether the car smashed into the tree, or whether it was the tree that hit the car (Cellier et al., 1995). In simulation, we don't know whether a resistor is a "voltage-drop causer" ($V = R \cdot i$) or a "current-flow causer" ($i = V/R$). Physically, there is only one type of resistor; what we need inside a state-space model has to do with the peculiarities of the chosen solution technique and not with the physical foundations of the equation. Physical equations should be declarative in nature.

A decent modeling system should allow a user to specify the model in terms of physical conservation laws rather than in the derived state-space form. Dymola does precisely that. In addition to "vertical equation sorting" (a feature offered by most simulation software), Dymola also offers "horizontal equation sorting," i.e., the Dymola compiler makes a structural analysis of all equations in order to determine what variable to solve for from each equation and provides a solution. This makes Dymola a truly modular modeling system, much more so than block-diagram languages.‡

---

*There meanwhile exists a Symbolic Toolbox interfacing MATLAB with MAPLE V.

†There exists a second tool called Omola (Andersson, 1994) offering features similar to those available in Dymola. However, Omola is still a university prototype.

‡Support of a differential algebraic equation solver (e.g., DASSL) at run time, as Xmath does, is an alternative approach that overcomes the causality assignment problem at least in some cases.

While this feature is certainly a very important asset of an object-oriented modeling system, it also revolutionizes the way, control-system design can be accomplished. Whereas in simulation, the inputs and the model parameters are known and the outputs are computed, control-system design uses exactly the same model, but declares inputs and outputs as known and the controller parameters as unknown. If the controller parameter happens to be the value of a potentiometer, Dymola would then simply turn the above equation into $R = u/i$.

Control-system design implicitly solves the problem of plant inversion. Given the desired system outputs, if only we knew the inverse plant dynamics, we could calculate the necessary plant input (controller output) easily. Evidently, if the plant itself is strictly proper (has no direct input/output coupling), then the inverse plant would be nonproper. To overcome this difficulty, we may need a reference model with sufficiently more poles than zeros, such that the cascade model of the reference model and the inverse plant model is at least proper. In Mugica and Cellier, 1994, it was shown by means of a highly nonlinear tanker ship model how Dymola can tackle this problem by simply connecting the output of the plant model to the output of the reference model and declaring the input of the plant as the desired output. In general, such a model will be a higher-index differential algebraic equation (DAE) model (Brenan et al., 1989), but Dymola has no problem reducing such a higher-index DAE model to state-space form (Cellier and Elmqvist, 1993). Dymola then generates a traditional simulation program in any one among a variety of different formats, including ACSL, Simnon, Desire, the textual (S-function) SIMULINK format, as well as plain Fortran or C.

It is believed that the future in CACSD lies in mixed symbolic and numerical formula and data processing, and Dymola offers a highly sophisticated, effective, and efficient tool to do so.

Dymola comes with a graphical front end of its own called Dymodraw. Dymodraw replaces the earlier HIBLIZ system. It offers visual object-oriented modeling of physical processes. Models of physical objects are represented through icons with interface points. Two objects can be connected through their interface ports by drawing a line between two ports of the two objects. An icon editor can be invoked to encapsulate an entire topological layer into a hierarchically higher-level object. Note that this is fundamentally different from a block-diagram editor, since connections between objects are nondirectional. They simply denote that two objects share some common variables, but do not impose any constraints on the direction of information flow between these objects. For example, an electrical circuit diagram is a special instance of an object diagram. In this example, every connection represents a wire connecting two circuit elements, making them share two variables, namely the electrical potential and the current flowing through the wire.

Figure 1 shows a screen dump of a Dymodraw session. The top center window is an object diagram window used to describe the electronic control circuitry driving a dc motor, which in turn drives the gear train of a robot joint. The control logic used to drive the electronics is shown in the bottom center window. This is a more traditional block-diagram window computing from the reference position qr and its derivative qr' together with the actual position q and its derivative q' the current ir for driving the motor. The overall robot model is shown in the right window. It is a six-degree-of-freedom robot arm. Six separate instantiations of the control, electronics, dc motor, and gear train modules are used to interface with the six limbs of the robot. To the left, some other windows are shown containing icons representing library modules, as well as some tools.

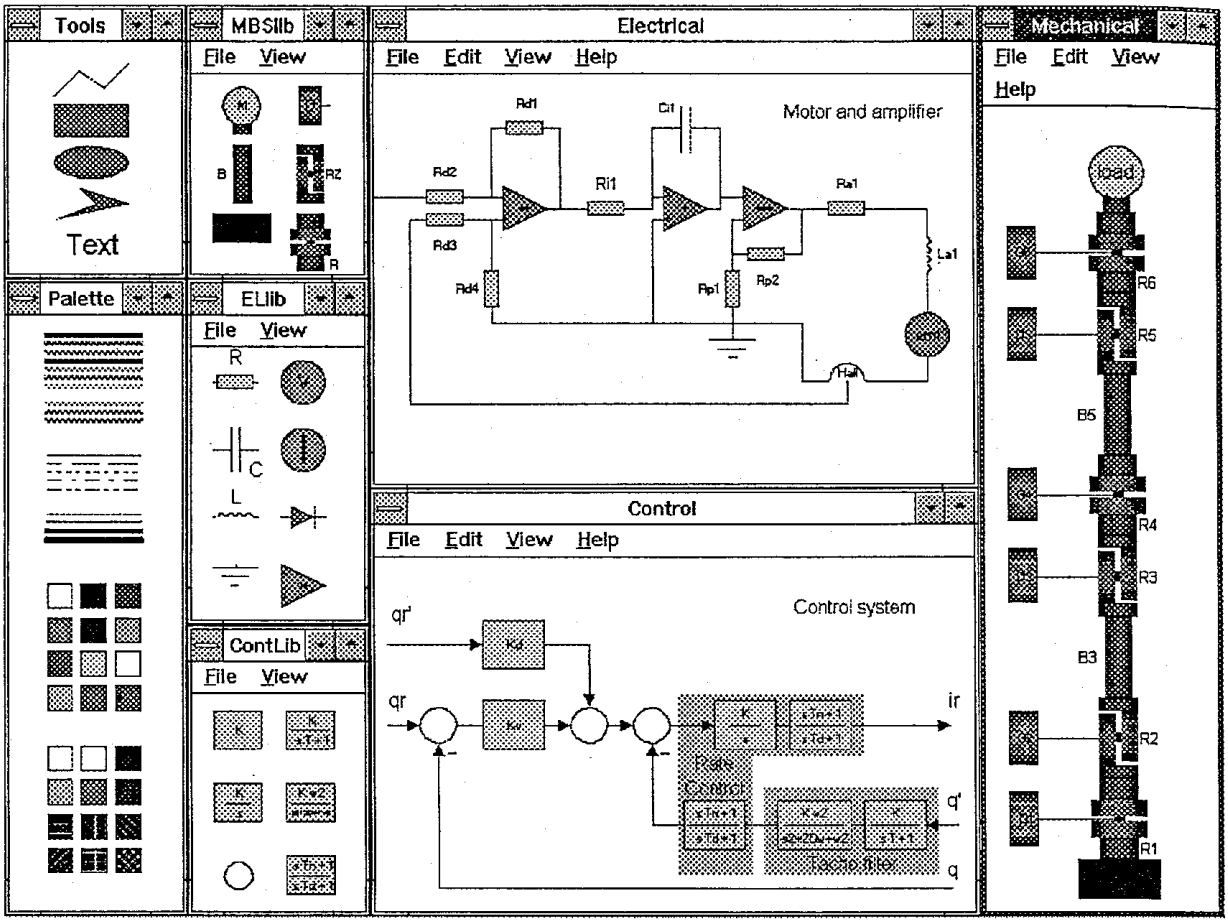

Figure 10.1 Dymodraw screen dump.

Dymodraw has made programs such as SIMULINK and SystemBuild perhaps less popular. However, Dymodraw is not yet as refined and user-friendly as its two competitors. Note that what is wrong with SIMULINK and SystemBuild is not their graphics processors; they are fine. What need to be addressed are the underlying graphics compilers that are (assignment) statement-oriented rather than (declarative) equation-oriented. Dymodraw is a by now fairly standard object-oriented graphics processor due to its underlying symbolic formula manipulator, i.e., Dymola.

10.5 STANDARDIZATION VERSUS DIVERSIFICATION

The matrix notation of MATLAB-like languages is so natural that we do not see a need for any other notation in this respect. Although the division operators / and \ for right and left division are not "standard" operators in the classical mathematical sense, after MATLAB became available and popular, even theoretical papers started using this notation for simplicity. It is hoped that a MATLAB-like notation will also be introduced into CSSLs as an additional tool for the description of state-space models as this has already happened in Dymola.

In IMPACT, we used additional operators for a third dimension, thus operating effectively on complex tensors in place of complex matrices. Multivariable systems can be expressed in terms of polynomial matrices in which each matrix element may be a polynomial in the linear operator s (or z in the discrete case). We introduced the ^ operator to separate polynomial coefficients and (alternatively) the | operator to separate polynomial roots. Thus, the polynomial matrix

$$\mathbf{P} = \begin{bmatrix} (3s^2 + 10s + 3) & (2s - 3) \\ s^3 & (-s^2 - 7s - 10) \end{bmatrix} \tag{10.23}$$

can, in IMPACT, be coded as

```
P = [[3^10^3],   [-3^2]
     [^^^1],     [-10^-7^-1]]
```
(10.24)

or alternatively as

```
PF = [3*[-3|-(1/3)],   2*[|1.5]
      [0|0|0],         -1*[-2|-5]]
```
(10.25)

to denote the factorized form

$$\mathbf{PF} = \begin{bmatrix} 3(s + 3)(s + 1/3) & 2(s - 1.5) \\ s*s*s & -(s + 2)(s + 5) \end{bmatrix} \tag{10.26}$$

The two operators ^ and | naturally extend the previously introduced operators "," (used to separate matrix columns) and ";" (used to separate matrix rows). Once selected, the data representation is maintained until the user decides to convert the polynomial matrices into another data representation, for example, by writing **PF**=FACTOR(**P**) or **P**=DEFACTOR(**PF**). Factorized polynomial matrices and defactorized polynomial matrices are two different data structures in IMPACT. Note that FACTOR(**PF**) results in an error message. This notation has meanwhile been adopted by the developers of M as well (Gavel and Herget, 1984).

Of course, it is natural to define once and forever

```
s = [^1]
```
(10.27)

(which in fact is an IMPACT system variable). Thus, one can also write a polynomial as

```
p1 = 3*s**2 + 10*s + 3
```
(10.28)

or alternatively as

```
p1 = 3*(s + 3)*(s + (1/3))
```
(10.29)

which nevertheless, in both cases, results in a polynomial of type defactorized polynomial, because the s operator was coded in a defactorized form. To prevent this from happening, the user could write

```
sf = [|0]
```
(10.30)

and thereafter

```
p1f = 3*(sf + 3)*(sf + (1/3))
```
(10.31)

which is not recommended, however, because frequent defactorizations and refactorizations take place in this case. Note the consequent overloading of the + and * operators in these examples. Depending on the types of operands, a different algorithm is employed to perform the operation.

Also, with respect to the embedded procedural language, an informal standard can be achieved. The procedural language of MATLAB, for instance, is very powerful. It basically extends the PASCAL programming style, operating conveniently on the new matrix data structures. Very useful, for instance, is the extension of the PASCAL-like "for" statement:

```
for I = [1, 3, 7, 28],
    This "for" loop shall be executed precisely four times with I      (10.32)
    = 1, I = 3, I = 7, and I = 28, respectively.
```

IMPACT employs an ADA style instead of the previously advocated PASCAL style. It actually does not matter too much which style is adopted in a forthcoming standard, but any standard would be highly welcome to allow smooth exchange of the extensive available soft-coded macro libraries. There is really no good reason to stick to the prevalent variety of only marginally different procedural languages.

Also with respect to the user interface, de facto "pseudostandards" have already been established. Window interfaces look more and more similar to the Macintosh interface. (Although the Macintosh was not the first machine to introduce windowing mechanisms, it was this machine that made this new technique popular.) The mouse is a very convenient, flexible, and fast-input device and has made the previously fashionable crosshair cursors and light pens obsolete; crosshair cursors are both uncomfortable to use and slow, and light pens demand very expensive screen sensors. However, there are mice with one, two, or three buttons. Any standard would be equally acceptable, but a standard must be found. Once the fingers are used to one system, it is hard to adjust to another.

With respect to the actual functions offered, we shall probably not see a standard quickly. The current diversification into different application areas and design methodologies is most likely to be around for some time, and we actually welcome this: too early a standard can freeze the lines and hamper the introduction of innovative new concepts.

Logically related functions should be combinable into libraries. Whereas CTRL-C offers a library facility, it is not flexible enough. In particular, it is unfortunate that user functions in CTRL-C cannot be accompanied by help text to be included in the standard CTRL-C help facility. Also, the necessity to define functions before their use is inconvenient: one changes CTRL-C functions quite frequently. If the altered functions are not immediately redefined, CTRL-C still uses the previous version, and the user waits for the introduced modification to have the desired effect. MATLAB's toolbox concept is much more powerful and should become a standard.

Another interface, which is rarely even noticed by the casual CACSD software user, is the interface to a database in which results of computations, as well as programming modules, notebook files, and so on, may be stored. To promote the state of the art of CACSD software further, it is imperative that a database interface standard be defined. Lacking such a standard, most current CACSD software developers do not even offer a

database interface but rely fully on the file-handling mechanism (directory structure) of the embedding operating environment. This mechanism is computationally efficient (the record manager, on every computer, is strongly optimized to suit the underlying hardware), but the mechanism is entirely insufficient for our task. The immediate effect of the lack of an appropriate database concept is a jungle of small and smallest data and program files scattered over different subdirectories, which makes it hard to retrieve data and programs that were previously stored for later reuse. As an example, a particular **A** matrix of a linear system is probably not related to the problem under investigation at all but is stored as a nonmnemonic file A.DAT located somewhere in the directory structure of the underlying operating system. Little has been done to address this pertinent problem. Probably most advanced in this context is the work of Maciejowski (1984) and Maciejowski and Szymkat (1994).

An IFAC working group discussing guidelines for CACSD software was generated, which consists of three subgroups for the discussion of the following:

1. CACSD program interfaces (including graphics)
2. CACSD program data exchange
3. CACSD program algorithm exchange

A similar IEEE working group exists as well. It is hoped that these two bodies will be able to promote a forthcoming CACSD standard (Maciejowski and Taylor, 1994).

10.6 SIMULATION AND CACSD

Let us discuss next how especially linear system simulation has been implemented in some of the current CACSD programs:

In CTRL-C, there is a simulation function that takes the following form:

$$[\mathbf{y}, \mathbf{x}] = \text{simu}(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{u}, \mathbf{t}) \tag{10.33}$$

where **a**, **b**, **c**, and **d** are the system matrices describing a linear continuous-time MIMO system, **t** is a time base (that is, **t** is a vector of time instants), **u** is the input vector sampled over the time base (that is, **u** is actually a matrix; each row denotes one input variable, and each column denotes one time instant), **y** is the output vector (that is, **y** is a matrix with rows denoting output variables and columns denoting time instants), and **x** is the state vector (which is also a matrix with according definitions). Initial conditions can be specified in a previous call of the same function:

simu('IC',$\mathbf{x}_0$)

and the integration method can also be declared in a similar manner:

simu('ADAMS',relerr,abserr,maxstp)

An equivalent function dsimu exists for discrete-time systems. The system matrices can be constructed from subsystem descriptions by a series of interconnection functions (series, parallel, interc, and minreal).

In IMPACT, we chose a slightly different approach. Because systems and trajectories are identifiable as separate data structures, we can once again overload the meaning of the primitive operators. Time bases ("domains") are created by means of the functions

lindom and logdom and/or by use of the & operator (concatenation operator):

$$\mathbf{t} = \text{lindom}(0.,1.,0.1)\ \&\ \text{lindom}(2.,20.,1.)\ \&\ 50.\ \&\ 100 \tag{10.34}$$

which generates a domain consisting of 23 points: [0., 0.1, 0.2, . . . , 0.9; 1., 2., 3., . . . , 19., 20, 50., 100.]. Trajectories are functions over domains; thus,

$$\mathbf{u} = [\sin(\mathbf{t});\ \cos(\mathbf{t})] \tag{10.35}$$

which creates a trajectory column vector **u** evaluated over the previously defined domain **t**. Linear systems are generated by the lincont and lindisc functions:

$$\mathbf{s}_1 = \text{lincont}(\mathbf{a}_1,\mathbf{b}_1,\mathbf{c}_1,\mathbf{D}=>d1,\mathbf{x}_0=>[0.5;2.;-3.7]) \tag{10.36}$$

The three matrices $\mathbf{a}_1$, $\mathbf{b}_1$, and $\mathbf{c}_1$ are compulsory positional parameters, whereas the input-output matrix (**D**) and the initial condition vector ($\mathbf{x}_0$) are optional (defaulted) named parameters.

Series connection between two sybsystems is expressed as $\mathbf{s}_2{*}\mathbf{s}_1$, that is, multiplication in reverse order (exactly what it would be if the two subsystems were expressed through two transfer function matrices $\mathbf{g}_2{*}\mathbf{g}_1$); parallel connection is expressed by the + operator, and feedback is expressed by the \\ operator:

$$\mathbf{g}_{\text{tot}} = \mathbf{g} \setminus\setminus (-\mathbf{h}) \tag{10.37}$$

(**g** fed back with $-\mathbf{h}$), independently of whether **g** and **h** are expressed as transfer function matrices or as linear system descriptions. Simulations finally are expressed by overloading the * operator once more:

$$\mathbf{y} = \mathbf{s}_1{*}\mathbf{u} \tag{10.38}$$

which simulates the system $\mathbf{s}_1$ (which in our example must have two inputs) from 0. to time 100., interpolating between the specified values of the input trajectory vector **u**, and sampling the output trajectory vector **y** over the same domain. Thus,

$$\mathbf{t}_{\text{out}} = (\mathbf{s}_2{*}\mathbf{s}_1){*}\mathbf{t}_{\text{in}} \tag{10.39}$$

series connects the two subsystems $\mathbf{s}_1$ and $\mathbf{s}_2$ and then performs one simulation over the combined system. On the other hand,

$$\mathbf{t}_{\text{out}} = \mathbf{s}_2{*}(\mathbf{s}_1{*}\mathbf{t}_{\text{in}}) \tag{10.40}$$

simulates the subsystem $\mathbf{s}_1$ using $\mathbf{t}_{\text{in}}$ as input, samples the resulting output trajectory over the same domain, and then simulates the subsystem $\mathbf{s}_2$ using the previous result as an input by reinterpolating it between its supporting values. Of course, numerically the results of these two operations are slightly different, but conceptually, the associative law of multiplication holds.

In standard CSSLs, simulation is always viewed as the execution of a special-purpose program (the simulation program) producing simulation results (mostly in the form of a result file). There, the simulation program is viewed as the central part of the undertaking. No wonder such a concept does not lend itself easily to embedding into a larger whole in which simulation is only one task among many.

In CTRL-C, simulation is viewed as a function mapping an input vector (or matrix) into an output vector (or matrix). Clearly, simulation is here only one function among many others that can be performed on the same data.

In IMPACT, finally, simulation is viewed as a binary operator that maps two different data structures, namely, one of type system description (eventually also nonlinear) and the other type of trajectory into another data structure of type trajectory.

Of course, all three descriptions mean ultimately the same thing, yet the accents are drastically different. To prove our case, the reader versed in the use of one or the other of the CSSLs may try to code the IMPACT statement $\mathbf{t}_{out} = \mathbf{s}_2*(\mathbf{s}_1*\mathbf{t}_{in})$ as a CSSL simulation program. In most CSSLs, this simply cannot be done. The task would require two separate programs to be executed one after the other. The output from the first simulation run (implementing $\mathbf{t}_{aux} = \mathbf{s}_1*\mathbf{t}_{in}$) would have to be manually edited into a "tabular function" and used by the second simulation run (implementing $\mathbf{t}_{out} = \mathbf{s}_2*\mathbf{t}_{aux}$).

To give another example, when solving a finite-time Riccati differential equation, one common approach is to integrate the Riccati equation backward in time from the final time t_f to initial time t_0, because the "initial condition" of the Riccati equation is stated as $\mathbf{K}(t = t_f) = \mathbf{0}$ and because the Riccati equation is numerically stable in the backward direction only. The solution $\mathbf{K}(t)$ is stored away during this simulation and then reused (in reversed order) during the subsequent forward integration of the state equations with given $\mathbf{x}(t = t_0)$. Some of the available CSSLs allow solving this problem (mostly in a very indirect manner); others simply cannot be used at all to tackle this problem.

How can one handle this problem in CTRL-C? The first simulation is nonlinear (and autonomous), and the second is linear (and input dependent) but time varying; thus, we cannot use the simu function in either case. CTRL-C provides for a second means of simulation, though. A graphical model builder called Model-C (Systems Control Technology, 1990) and also an interface to the well-known simulation language ACSL were introduced. These interfaces allow making use of the modeling and simulation power of a full-fledged simulation language, and one is still able to control the experiment from within the more flexible environment of the CACSD program. Several of the CACSD programs discussed here follow this path, and it might indeed be a good answer to our problem if the two languages that are combined in such a manner are sufficiently compatible with each other and if the interface between them is not too slow. Unfortunately, this is currently not yet the case with any of the CACSD programs that use this route.

Let us illustrate the problems. We start by writing an ACSL program that implements the matrix Riccati differential equation

$$\frac{d\mathbf{K}}{dt} = -\mathbf{Q} + \mathbf{K}*\mathbf{B}*\mathbf{R}^{-1}*\mathbf{B}'*\mathbf{K} - \mathbf{K}*\mathbf{A} - \mathbf{A}'*\mathbf{K} \qquad \mathbf{K}(t_f) = 0 \tag{10.41}$$

Because ACSL does not provide for a powerful matrix environment, we must separate this compact matrix differential equation into its component equations. [ACSL provides a vector integration function, and matrix operations, such as multiplication and addition, could be (user-)coded by use of the ACSL MACRO language. However, this is a slow and inconvenient replacement for the matrix manipulation power offered in such languages as CTRL-C.] Furthermore, because ACSL does not handle the case $t_f < t_0$, we must substitute t by

$$t^* = t_f - t_0 - t$$

and integrate the substituted Riccati equation

$$\frac{d\mathbf{K}}{dt^*} = \mathbf{Q} - \mathbf{K}^*\mathbf{B}^*\mathbf{R}^{-1}{}^*\mathbf{B}'^*\mathbf{K} + \mathbf{K}^*\mathbf{A} + \mathbf{A}'\mathbf{K} \qquad \mathbf{K}(0) = 0 \tag{10.42}$$

forward in time from $t^* = 0$ to $t^* = t_f - t_0$. Through the interface (A2CLIST), we export the resulting $\mathbf{K}_{ij}(t^*)$ back into CTRL-C, where they take the form of ordinary CTRL-C vectors. Also in CTRL-C, we must manipulate the components of $\mathbf{K}(t)$ individually, because $\mathbf{K}(t)$ is a trajectory matrix, that is, a three-dimensional structure. However, CTRL-C handles only one-dimensional structures (vectors) and two-dimensional structures (matrices), but not three-dimensional structures (tensors). Backsubstitution can be achieved conveniently in CTRL-C simply by reversing the order of the components of each of the vectors, as follows:

$$\begin{aligned} [n,m] &= \text{size}(\mathbf{K}_{ij}) \\ nm &= n^*m \\ \mathbf{K}_{ij} &= \mathbf{K}_{ij}(nm{:}-1{:}1) \end{aligned} \tag{10.43}$$

Now, we can set up the second simulation:

$$\frac{d\mathbf{x}}{dt} = [\mathbf{A} - \mathbf{K}(t)^*\mathbf{B}]^*\mathbf{x} \qquad \mathbf{x}(t_0) = \mathbf{x}_0 \tag{10.44}$$

What we would like to do is to ship the reversed $\mathbf{K}_{ij}(t)$ back through the interface (C2ALIST) into ACSL and use them as driving functions for the simulation. Unfortunately, ACSL is not (yet!) powerful enough to allow us to do so. Contrary to the much older CSMP-III system, ACSL does not offer a dynamic table load function (CALL TVLOAD). Thus, once the $\mathbf{K}_{ij}(t)$ functions have been sent back through the interface into ACSL, they are no longer trajectories, but simply arrays, and we are forced to write our own interpolation routine to find the appropriate value of $\mathbf{K}$ for any given time t. After all, the combined CTRL-C/ACSL software is indeed capable of solving the posed problem, but not in a very convenient manner. This is basically because ACSL is not (yet!) sufficiently powerful for our task and the interface between the two languages is still awkward. Because of the weak coupling between the two software systems, it might indeed have been easier to program the entire task in ACSL alone, although this would have meant doing without any of the matrix manipulation power offered in CTRL-C.

What about IMPACT? in IMPACT, it was decided not to rely on any existing simulation language but rather to build simulation capabilities into the CACSD program itself. This is partly because (as the preceding example shows) the currently available simulation languages are really not very well suited for our task and partly a result of our decision to employ ADA as implementation language. Currently no CSSL has been programmed in ADA, so that we would have had to rely on the "pragma concept" (which is the ADA way to establish links to software coded in a different language). However, we tried to limit the use of the pragma concept as much as possible because this feature does not belong to the standardized ADA kernel (and thus may be implementation dependent).

Until now, only the use of linear systems in IMPACT has been demonstrated. However, nonlinear systems can be coded as special macros (called system macros). The two

linear system types (lincont and lindisc) are, in fact, special cases of system macros. The Riccati equation can be coded as follows:

```
SYSTEM ricc_eq(a,b,q,rb) RETURN k IS
k = zero(a);
BEGIN
    k` = -q + k*b*rb*k - k*a - a'*k;
END ricc_eq
```

The state equations can be coded as

```
SYSTEM sys_eq(a,b,rb,x0) INPUT k RETURN x IS
x = x0
BEGIN
    x` = (a - rb*k)*x;
END sys_eq
```

The total experiment can be expressed in another macro (of type FUNCTION MACRO):

```
FUNCTION fin_tim_ricc(a,b,q,r,xbeg,time_base) IS
BEGIN
    back_time = REVERSE(time_base);
    rb = r\b';
    k1 = ric_eq(a,b,q,rb)*back_time;
    k2 - REVERSE(k1);
    x = sys_eq(a,b,rb,xbeg)*k2;
    RETURN <x,k2>;
END fin_tim_ricc;
```

Note the difference in the call of the two simulations. The first system (ricc_eq) is autonomous. Therefore, simulation can no longer be expressed as a multiplication of a system macro with a (nonexistent) input-trajectory vector. Instead, the system macro here is multiplied directly by the domain variable, that is, the time base. The second system, on the other hand, is input dependent. Therefore, the multiplication is done (as in the previously discussed linear systems) with the input trajectory. FIN_TIM_RICC can now be called just like any of the standard IMPACT functions (even nested). The result of this operation are two variables, **y** and **K**, of the trajectory vector and trajectory matrix type, respectively.

```
x0 = [0;0]; a = [0,1;-2,-3]; b = [0;1];
q = [10,0;0;100]; r = 1;
forw_time = LINDOM(0,10,0.1,METHOD=>'ADAMS',ABSERR=>0.001);
[y,k] = fin_tim_ricc(a,b,q,r,x0,forw_time);
plot(y)
```

As can be seen from this example, the entire integration information in IMPACT is stored in the domain variables, which makes sense because these variables contain part of the runtime information (namely, the communication points and the final time anyway. More-

over, this gives us a neat way to differentiate clearly between the model description, on the one hand, and the experiment description, on the other.

Obviously, this is a much more powerful tool for our demonstration task than even the combined ACSL/CTRL-C software. Unfortunately, contrary to CTRL-C, IMPACT has never been released. IMPACT, being a university prototype could simply not compete with professionally developed and maintained industrial products such as MATLAB or $MATRIX_X$. Even CTRL-C doesn't seem to carry a significant portion of customer base of matrix manipulation tools any longer.

10.7 OUTLOOK

How will the field of CACSD develop further over the next decade or so? To understand where we are heading, we need to assess where we currently stand. In the past, and this still holds for the first generation of CACSD tools, the application programmer was talking about program development. A program is a tool that calculates something in a sequential manner when executed on a digital computer. Some programs were parameterized, that is, accepted input data to determine partly what was to be calculated. The major emphasis was on the program, whereas the data were of relatively minor importance. There was a clear distinction between the program (a piece of static code in memory) and the data (a portion of memory that changed its content during execution of the program).

With the new generation of CACSD tools, we departed from this viewpoint drastically. New CACSD programs are in themselves true programming languages; that is, the application programmer no longer relies on the computer manufacturer to provide the languages to be used but creates his or her own special-purpose languages. The difference is simply that less and less of the computational task is frozen in code, and more and more of it is parameterized, that is, data driven. The data in itself reached such a degree of complexity that their appropriate organization became essential. The user interface, previously an unimportant detail, turned into a central question that decided whether a particular CACSD tool was good or bad, even more than the algorithmic richness provided within the program. What we gained by this change in accent was a dramatic increase in flexibility offered by the CACSD tools; what had to be paid in return was a certain decrease in runtime efficiency. However, with the advent of more powerful computers (an engineering workstation of today compares in number-crunching power easily with a mainframe computer of no more than a decade ago), this sacrifice could be gladly made. Moreover, it was often true that the compilation and linkage of a simulation program took 10 times longer than the actual execution of the program (at least for sufficiently simple applications). With the advent of the direct-executing (that is, fully data driven) simulation languages, such as SIMNON (Elmqvist, 1975, 1977), DESCTOP, and DESIRE (Korn, 1985, 1987, 1989), one can obtain simulation results immediately, and even if the simulation program executes 50% slower than it would if it were properly compiled, the increased flexibility of the tool (ease of model change) pays off easily even with respect to the total time spent at the computer terminal. This is useful within a CACSD environment, especially for relatively small simulation models. This is also the route that The MathWorks took with SIMULINK and that ISI took with SystemBuild. For larger models, interactivity is less important than modularity, and this is where Dymola fits in. Dymola has already been integrated with MATLAB, since Dymola can either

generate (textual) SIMULINK code, or directly C code to be simulated using Dymosim (Otter, 1995), which develops its output in the form of MATLAB data files to be imported easily and conveniently into MATLAB for further processing and/or graphing.

However, we are currently at the edge of taking yet another step. We have already gotten used to multiwindow user interfaces, to object-oriented programming style, to language-sensitive editors, to CAD databases, and so on. Are these really issues that can (or should) be tackled at the level of a programming language? Are these not rather topics to be discussed at the level of the underlying operating systems? If we say that we need a CAD database to store our models and resulting data files, do we not simply express that the file storage and retrieval system of the operating system in which the tool is being embedded is not powerful enough for our task? Are not interactive languages, such as MATLAB and DESCTOP, (very primitive) special-purpose operating systems in themselves? We indeed believe that future programming systems will blur the previously clear-cut distinction between programming languages and the operating systems in which they are embedded. This problem was realized by the developers of ADA, who understood that a complex tool, such as ADA, cannot be designed as a programming language with a clean interface to the outside, implementable independently of the operating system under which it is to run. Instead, its developers considered an ADA environment to be offered together with the ADA language. The ADA environment is basically nothing but a (partial) specification of the operating system in which the ADA language is to be embedded. The same is true with respect to CACSD tools. In IMPACT, we were not yet able to address this question in full depth: the ADA environment itself was not yet completely defined and we would like to borrow as much as possible from ADA concepts. In M (Gavel and Herget, 1984), this question was addressed and led to the development of yet another tool, EAGLES (Lawver, 1985), an object-oriented, multitasking, multiwindowing operating system, under which M is to run. The import/export of M variables between different sessions (windows) is not programmed in M itself but is supported by EAGLES. The entire graphics system is a facility provided by the EAGLES operating system rather than being implemented as an M tool. EAGLES operates on a rather involved database that serves as a buffer for all data to be shuffled back and forth between the different tools (such as M) and the operating system EAGLES itself.

One way to overcome the previously mentioned problems may be to standardize the operating system itself. The UNIX operating system presents one step in this direction. There already exist a large number of UNIX implementations for various computers. The idea is splendid. UNIX has already largely replaced older operating systems such as VMS; the reason for its success is its open architecture. UNIX is vendor-independent and completely open to the system programmer and offers the ultimate in flexibility to the system programmer. Unfortunately, the original UNIX kernel (the UNIX "standard," so to speak) was too small, and there is therefore unnecessary and unjustified diversity in UNIX implementations.

What about new facilities offered in future CACSD tools? We expect to see more and more flexibility with respect to the data interface. The ultimate data-driven programming is a language in which there is essentially no longer a difference between code and data. Each operation that can be performed in the language is itself expressed as an entry in a database and can thus be altered at any moment.

One such environment is LISP. Basically, the only primitive operations in LISP are addition and removal of entries from lists. These operations are themselves expressed as

entries in lists. When interpreted as operation, the first entry in the list is the operator, and all further entries are its parameters. For these reasons, LISP programs exhibit a serious runtime inefficiency. A numeric algorithm implemented in LISP will probably execute two to three orders of magnitude more slowly than the same algorithm implemented in a conventionally compiled language. Moreover, LISP is often rather unwieldy with respect to how a particular numeric algorithm must be specified. However, LISP certainly also presents the ultimate in flexibility. Suddenly, self-modifying code has become a feasibility and can be employed to achieve amazing results. Moreover, in LISP, numeric data are entries in lists just like any other data. Thus, nonnumeric data processing is as efficient as numeric data processing, and in this arena, LISP competes a little more favorably with conventional programming techniques. Also, steps have been taken to alleviate some of this inherent inefficiency. Incremental compilers in place of pure interpreters can increase the runtime efficiency by roughly one order of magnitude. Furthermore, a LISP interpreter is an extremely simple program compared with a conventional compiler. A (basic) LISP interpreter can be coded in roughly 600 lines of (LISP) code. Owing to this simplicity, it may make sense to implement part of this task in hardware rather than in software. The machine instructions of a special-purpose LISP machine can be tuned to optimize the efficiency of executing LISP primitives. Such machines have already become available and have helped overcome at least part of the inefficiency of LISP. Unfortunately, they have disappeared again from the market about as fast as they had appeared before, since few were willing to code all their programs in LISP.

With respect to the user interface, many of the difficulties of LISP can be avoided by changing the world view once more. LISP is basically process oriented, but PROLOG is activity oriented. That is, in LISP, the programmer takes the standpoint of the operator (What do I do next with my data?), whereas in PROLOG, the programmer takes the standpoint of the data (What needs to happen to me next?). This helps to concentrate activities to be performed into one piece of code rather than having them spread all over. Unfortunately, digital computers are still sequential machines, whereas activity programming is not procedural in nature. As a consequence, PROLOG is expected to be more inefficient even than LISP. (However, PROLOG can rather easily be implemented in LISP, and thus, there also exist PROLOG environments on LISP machines, and they function amazingly well.) PROLOG primitives are more compound than LISP primitives. The natural consequences of this enhanced degree of specialization are shorter and better readable PROLOG programs, on the one hand, but less flexibility, on the other. Not every program that can be conceived in LISP can easily be implemented in PROLOG, whereas PROLOG programs can be implemented in LISP.

An area that will be boosted by such concepts as advertised in PROLOG and LISP is the integration of CACSD software with expert systems. Expert systems are programs that evaluate a set of parameterized rules (conditional statements with mostly nonnumeric operands) by plugging in appropriate parameter values. The set of available parameter values is called the knowledge of the expert system. Each evaluation may generate new knowledge, and eventually even new rules. To accommodate this new knowledge (new rules), the rules of the expert system are evaluated recursively until no further facts (knowledge) can be derived from the current state of the program.

Why is it that many computer experts smile at the current efforts in expert system technology? To design an expert system, one needs expert knowledge. For this reason, most of the early expert systems were written by experts in the application area rather

than by experts in the implementation tool. Such programs did not always exploit the latest in software technology. Expert systems are thus often envisaged as question-and-answer–driven programs with very limited capabilities. However, our definition of the term "expert system" did not mention the user interface at all. In fact, the user interface (that is, the port through which new knowledge is entered into the knowledge base of the expert system) is completely decoupled from the mechanisms of rule evaluations (the inference engine) and can be any of the previously mentioned interface types (question and answer, command, menu, form, graphic, and window interface).

Indeed, have not expert systems been further developed than what most people think? Is not MATLAB in fact an expert system for linear algebra? Is not every single CACSD tool an expert system for control system design? They surely exhibit all properties of expert systems. To prove our case, let us examine the MATLAB statement

$$x = b/a \tag{10.45}$$

a little more closely. Certainly, the interpreter of this statement performs symbolic processing. Once it has determined the type of operation to be performed (division), it checks the types of the operands. If a is a scalar, all elements of b must be divided by a. If a is a square matrix, a Gaussian elimination must take place to determine x. Finally, if a is a rectangular matrix, x is evaluated as the solution of an over- or under-determined set of equations in a least-squares sense. Quite obviously, these are rules to be evaluated.

Of course, most people would not call MATLAB an expert system (and neither would we). However, more expert system technology is readily available than what is commonly exploited. For example, most expert systems today constantly perform operations on symbolic data. It is true that the data to be processed are input in a symbolic form. However, this does not mean that they must be processed within the expert system in a symbolic form as well. Compiler writers have known this for years. The scanner interprets the input text, maps tokens (symbols) into more conveniently processable integers, and stores them in fast addressable data structures. This process is called hashing. During the entire operation of compilation (and eventually also symbolic debugging), the system operates on these numeric quantities in place of the symbolic ones. Only upon output (e.g., for generating the cross-reference table), are the original symbols retrieved through the hash table. For example, Dymola performs symbolic formula manipulation with high speed due to appropriate data structures and efficient algorithms based on a bipartite graph representation of the model structure. Such mechanisms could easily be used in expert systems to increase their efficiency, but this is rarely done today. SAPS (Uyttenhove, 1979; Klir, 1985; Cellier and Yandell, 1987; Cellier et al., 1994), for instance, can be used for qualitative simulations of input/output models (that is, models described through sets of input and output trajectories rather than by a symbolic structure). The trajectories can be either discretized continuous variables or variables that are discrete in nature. Often, one would like to characterize a signal as being [*much_too_small, too_small, just_right, too_large*, or *much_too_large*]. These symbols are mapped into the set of integers [0, 1, 2, 3, and 4]. The authors of SAPS called this process recoding.

How can the emerging expert system technology be exploited by CACSD software? As a first step, the error-reporting facility, the help facility, and the tutorial facility of CACSD tools should be made dynamic. Today, such facilities exist in most CACSD programs, but they are static; that is, the amount and detail of information provided by the system are insensitive to the context from which it was triggered. The idea is quite

old. IBM interactive operating systems have offered for many years a two-level error-reporting facility. When an error occurs, a short (and often cryptic) message is displayed that may suffice for the expert but is inadequate for the novice. Thus, after receiving such a message, the user can type ? which is honored by a more detailed analysis of the problem. The query facility of IMPACT is another step in this direction. Another implementation has been described by Munro et al. (1986).

Also, Åström and Ljung are working on such a facility for IDPAC (private communication). The idea is the following: Rather than allowing students to queue in front of Karl Johan's office, his knowledge about the use of the IDPAC algorithms (when to use what module) should be coded into the program itself, providing the students with an adaptive tutorial facility for identification algorithms. Thus, a computer-aided instruction facility is being built into the CAD program. A similar approach has been proposed by Taylor and Frederick (1984).

Another related idea was expressed by Åström (private communication). That is, to add a command spy to his IDPAC software. Here, the idea is as follows: instead of waiting until the student realizes that he is doing something wrong, and therefore seeks the professor's advice, the professor stands, in a figurative sense, behind the student and watches over his shoulder to see what he is doing. As long as the student is doing fine, the professor (that is, the command spy) keeps quiet, but when the student tries to perform an operation that is potentially dangerous to the integrity of his data or that is likely to lead to illegitimate conclusions, the command spy becomes active and warns the student about the consequences of what he is doing.

A similar feature could be built into a language-sensitive editor. This would allow checking a CACSD program early not only for syntactic correctness but also for semantic correctness. Some of the semantic tests are, of course, data dependent, and these can be performed only at execution time.

Other improvements can be expected from screening data for automated selection of the most adequate algorithms. This is similar to the previously mentioned operator overloading facility, but here the algorithm is selected not on the basis of the types of the operands but on the basis of the data itself. As a typical example, we could mention the problem of inverting a matrix. Obviously, if the matrix is unitary, its inverse can be obtained by simply computing the conjugate complex transpose of the matrix, which is much faster and gives rise to less error accumulation than computation of the inverse by Gaussian elimination, for example. If the matrix is (block) diagonal, each diagonal block can be inverted independently. If the matrix presents itself in a staircase form, yet another simplified algorithm can be used, and so on. Thus, the matrix should be checked for particular structural properties, and the most appropriate algorithm should be selected on the basis of the outcome of this test. A good amount of knowledge about data classification algorithms exists, knowledge that is not being exploited by many of today's CACSD programs.

Finally, we expect that even new control algorithms will arise from expert system technology. Today's control algorithms are excellent for local control of subsystems. They are not as good for global assessment of complex systems. A complex system, such as a space station or a nuclear power plant, must be monitored, and expert system technology may be used to decide when something odd has happened or is about to happen. A global control strategy must then take over and decide what to do next. Currently, human operators do a much better job in this respect than automatic controllers. However, they do not solve Riccati equations in their heads. Instead, they decide on the basis of

qualitative, that is, highly discretized, information processed by use of a mental model of the process.

De Albornoz and Cellier (1993) investigated the use of inductive reasoning for monitoring a nuclear power station. A detailed 5000-page emergency procedure manual exists for any such plant. Thus, if something goes awry (a so-called transient occurs), the problem is not to figure out how to handle the situation: the problem is which page of the emergency procedure manual to read. Thus, the supervisory control system is supposed to do nothing but provide the operator with a (set of) page number(s) of the emergency procedure manual. Cellier et al. (1993) provide a picture of the major issues that need to be tackled in computer-aided design of intelligent controllers. This paper introduced the terminology "fault-tolerant controller" to mean a robust controller that watches over the integrity of the plant and either adjusts its control activities whenever it notices a qualitative change in plant behavior or, at least, initiates a graceful shutdown procedure and calls for help. The terminology "self-aware controller" was introduced to denote the capability of a controller to watch over its own sanity and find out if something has gone awry within the controller. An elaborate example of a self-aware control application is given in the paper. Finally, the terminology "cognizant controller" is introduced to mean the ability of a controller to assess ahead of time the effects of its control actions and base its decisions on a mental assessment of a mental simulation performed on a mental model of the plant to be controlled in faster than real time. The proceedings of the most recent CACSD conference (Mattsson et al., 1994), indicate that issues of intelligent controller design occupy a significant portion of reported activities.

In summary, CACSD is still a very active research field, as recent books on this topic show (Jamshidi and Herget, 1993; Linkens, 1993), and more results are expected. It is hoped that our survey and discussion may stimulate more research.

REFERENCES

Ackermann, J. (1980). Parameter space design of robust control systems, *IEEE Trans. Automatic Control, AC-25*, 1058–1072.

Agathoklis, P., Cellier, F. E., Djordjevic, M., Grepper, P. O., and Kraus, F. J. (1979). INTOPS, educational aspects of using computer-aided design in automatic control, in: *Proceedings of the IFAC Symposium on Computer-Aided Design of Control Systems*, Zürich, Switzerland, August 29–31, 1979, M. A. Cuénod, ed., Pergamon Press, Oxford, pp. 441–446.

Andersson, M. (1994). Object-oriented modeling and simulation of hybrid systems, Ph.D. dissertation LUTFD2/TFRT-1043-SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Aplevich, J. D. (1986). Waterloo control system design packages (WCDS and DSC), personal communication, Dept. of Electrical Engineering, University of Waterloo, Waterloo, Ontario, Canada.

Asada, H., and Slotine, J. J. E. (1986). *Robot Analysis and Control*, Wiley, New York.

Åström, K. J. (1980). Self-tuning regulators—design principles and applications, in *Applications of Adaptive Control*, K. S. Narendra and R. V. Monopoli, eds., Academic Press, New York, pp. 1–68.

Åström, K. J. (1985). Computer-aided tools for control system design, in: *Computer-Aided Control Systems Engineering*, M. Jamshidi and C. J. Herget, eds., North Holland, Amsterdam, pp. 3–40.

Athens, M., ed. (1978). On large scale systems and decentralized control, *IEEE Trans. Automatic Control, AC-23*, special issue.

Atherton, D. P., and Wadey, M. D. (1981). Computer-aided analysis and design of relay systems, in: *IFAC Symposium on CAD of Multivariable Technological Systems*, Pergamon Press, New York, pp. 355–360.

Atherton, D. P., McNamara, O. P., Wadey, M. D., and Goucem, A. (1986). SUNS: The Sussex University Nonlinear Control System Software, in: *Proceedings of the 3rd IFAC Symposium on Computer-Aided Design in Control and Engineering Systems* (CADCE '85), Copenhagen, July 31 to August 2, 1985, P. M. Larsen and N. E. Hansen, eds., Pergamon Press, Oxford, pp. 133–136.

Augustin, D. C., Fineberg, M. S., Johnson, B. B., Linebarger, R. N., Sanson, F. J., and Strauss, J. C. (1967). The SCi continuous system simulation language (CSSL), *Simulation, 9*, 281–303.

Barker, H. A. (1994). Open environments and object-oriented methods: the way forward in computer-aided control system design, Proceedings IEEE/IFAC Joint Symposium on Computer-Aided Control System Design, Tucson, AZ, pp. 3–12.

Bartolini, G., Casalino, G., Davoli, F., and Minciardi, R. (1983). A package for multivariable adaptive control, in: *Proceedings of the 3rd IFAC/IFIP Symposium on Software for Computer Control* (SOCOCO '82), Madrid, Spain, G. Ferrate and E. A. Puente, eds., Pergamon Press, Oxford, pp. 229–235.

Birdwell, J. D., Cockett, J. R. B., Heller, R., Rochelle, R. W., Lamb, A. J., Athans, M., and Hatfield, L. (1985). Expert systems techniques and future trends in a computer-based control system analysis and design environment, in: *Proceedings of the 3rd IFAC Symposium on Computer-Aided Design in Control and Engineering Systems* (CADCE '85), Copenhagen, July 31 to August 2, 1985, P. M. Larsen and N. E. Hansen, eds., Pergamon Press, Oxford, pp. 1–8.

Borrie, J. A. (1986). *Modern Control Systems*, Prentice-Hall, Englewood Cliffs, NJ.

Boyd, S. P., and Barratt, C. H. (1991). *Linear Controller Design, Limits of Performance*, Prentice-Hall, Englewood Cliffs, NJ.

Brenan, K. E., Campbell, S. L., and Petzold, L. R. (1989). *Numerical solution of initial-value problems in differential algebraic equations*, North-Holland, New York.

Buenz, D. (1986). CATPAC—Computer-aided techniques for process analysis and control, personal communication, Philips Forschungslaboratorium Hamburg, Hamburg, Germany

Cellier, F. E. (1986a). Enhanced run-time experiments in continuous system simulation languages, in: *Proceedings of the 1986 SCSC Multiconference*, F. E. Cellier, ed., SCS Publishing, San-Diego, CA, pp. 78–83.

Cellier, F. E. (1986b). Combined continuous/discrete simulation—applications, tools, and techniques, Invited Tutorial, in *Proceedings of the Winter Simulation Conference* (WSC '86), Washington, D.C., J. R. Wilson, J. O. Henriksen, and S. D. Roberts, eds., pp. 24–33.

Cellier, F. E. (1991). *Continuous System Modeling*, Springer-Verlag, New York.

Cellier, F. E. (1993). Integrated continuous-system modeling and simulation environments, in: *CAD for Control Systems*, D. A. Linkens, ed., Marcel Dekker, New York, pp. 1–29.

Cellier, F. E. (1995). *Continuous System Simulation*, Springer-Verlag, New York.

Cellier, F. E., and Elmqvist, H. (1993). Automated formula manipulation supports object-oriented continuous-system modeling, *Control Systems, 13*(2), 28–38.

Cellier, F. E., and Mugica, F. (1995). Inductive reasoning supports the design of fuzzy controllers, *J. Fuzzy Intelligent Systems*, accepted for publication.

Cellier, F. E., and Pan, Y. (1995). Fuzzy adaptive recurrent counterpropagation neural networks: A tool for efficient implementation of qualitative models of dynamic processes, *J. Systems Engineering*, accepted for publication.

Cellier, F. E., and Rimvall, M. (1983). Computer-aided control systems design, invited survey paper, in: *Proceedings of the European Simulation Conference* (ESC '83), Aachen, Germany, W. Ameling, ed., Springer-Verlag, New York, pp. 1–21.

Cellier, F. E., and Yandell, D. W. (1987). SAPS-II: A new implementation of the systems approach problem solver, *Int. J. General Systems, 13*(4), 307–322.

Cellier, F. E., Grepper, P. O., Rufer, D. F., and Tödtli, J. (1977). AUTLIB, automatic control library, educational aspects of development and application of a subprogram package for control, in: *Proceedings of the IFAC Symposium on Trends in Automatic Control Education*, Barcelona, Spain, March 30 to April 1, 1977, Pergamon Press, Oxford, pp. 151–159.

Cellier, F. E., Schooley, L. C., Sundareshan, M. K., and Zeigler, B. P. (1993). Computer-aided design of intelligent controllers: Challenge of the nineties, in: *Recent Advances in Computer-Aided Control Systems Engineering*, M. Jamshidi and C. J. Herget, eds., Elsevier, Amsterdam, pp. 53–80.

Cellier, F. E., Nebot, A., Mugica, F., and de Albornoz, A. (1995). Combined qualitative/quantitative simulation models of continuous-time processes using fuzzy inductive reasoning techniques, *Int. J. General Systems*, accepted for publication.

Cellier, F. E., Otter, M., and Elmqvist H. (1995). Bond graph modeling of variable structure systems, *Proceedings ICBGM '95, Second International SCS Conference on Bond Graph Modeling and Simulation*, Las Vegas, NV, F. E. Cellier and J. J. Granda, eds., SCS Publ., La Jolla, CA. pp. 49–55.

Chi, S., and Cellier, F. E. (1991). Numerical properties of trajectory representations of polynomial matrices, in: *Proceedings CADCS '91, Computer-Aided Design in Control Systems*, Swansea, Wales, H. A. Barker, ed., Pergamon Press, Oxford, pp. 173–177.

Chow, J. H., Bingulac, J. H., Javid, S. H., and Dowse, H. R. (1983). *User's Manual for L-A-S Language*, System-Dynamics and Control Group, General Electric, Schenectady, NY.

De Albornoz Bueno, A., and Cellier, F. E. (1993). Variable selection and sensor fusion in automatic hierarchical fault monitoring of large scale systems, in: *Proceedings QUARDET '93, Qualitative Reasoning and Decision Technologies*, Barcelona, Spain, June 16–18, 1993, N. Piera Carreté and M. G. Singh, eds., CIMNE, Barcelona, pp. 722–734.

Denham, M. J. (1984). Design issues for CACSD systems, *Proc. IEEE*, *72*(12), 1714–1723.

Dow Chemical Company (1990). *SimuSolv Manual*, Midland, MI.

Dynasim. (1995). *Dymodraw User's Manual*, Dynasim AB, Lund, Sweden.

Elmqvist, H. (1975). SIMNON—An Interactive Simulation Program for Nonlinear Systems—User's Manual, Report CODEN: LUTFD2/(TFRT-7502), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Elmqvist, H. (1977). SIMNON—An interactive simulation language for nonlinear systems, in: *Proceedings of the International Symposium SIMULATION '77*, Montreux, Switzerland, M. Hamza, ed., Acta Press, Anaheim, CA, pp. 85–90.

Elmqvist, H. (1978). A structured model language for large continuous systems, Ph.D. Thesis, Report: CODEN: LUTFD2/(TRFT-1015). Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Elmqvist, H. (1980). A structured model language for large continuous systems, *IMACS TC3 Newsletter, 10*.

Elmqvist, H. (1982). A graphical approach to documentation and implementation of control systems, in: *Proceedings of the 3rd IFAC/IFIP Symposium on Software for Computer Control (SOCOCO '82)*, Madrid, Spain, G. Ferrate and E. A. Puente, eds., Pergamon Press, Oxford, pp. 95–100.

Elmqvist, H. (1994). *Dymola User's Manual*, Dynasim AB, Lund, Sweden.

Elmqvist, H., and Mattsson, S. E. (1986). A simulator for dynamic systems using graphics and equations for modeling, in: *Proceedings of the 3rd Symposium on Computer-Aided Control System Design*, Washington, DC.

Elmqvist, H., Åström, K. J., Schönthal, T., and Wittenmark, B. (1990). *SIMNON—User's Guide for MS-DOS Computers*, SSPA Systems, Gothenburg, Sweden.

Elmqvist, H., Cellier, F. E., and Otter, M. (1993). Object-oriented modeling of hybrid systems, in: *Proceedings ESS '93, European Simulation Symposium*, Delft, Netherlands, A. Verbraeck and E. J. H. Kerckhoffs, eds., October 25–28, SCS Publ., La Jolla, CA. pp. xxxi-xli.

Evans, D. C. (1985). The art of visual simulation, keynote address, *Proceedings of the Winter Simulation Conference (WSC '85)*, San Francisco, Evans & Sutherland Computer Corp., IEEE Publishing, Piscataway, NJ.

Fleming, P. J. (1979). A CAD program for suboptimal linear regulators. in: *Proceedings of the IFAC Symposium on Computer-Aided Design of Control Systems*, Zürich, Switzerland, August 29–31, 1979, M. A. Cuénod, ed., Pergamon Press, Oxford, pp. 259–266.

Frederick, D. K. (1985). Software summaries, in: *Computer-Aided Control Systems Engineering*, M. Jamshidi and C. J. Herget, eds., North Holland, Amsterdam, pp. 349–384.

Gavel, D. T., and Herget, C. J. (1984). The M language—an interactive tool for manipulating matrices and matrix ordinary differential equations, International Report, Dynamics and Controls Group, Lawrence Livermore National Laboratory, University of California, Livermore, CA.

Golub, G. H., and Wilkinson, J. H. (1976). Ill-conditioned eigensystems and the computation of the Jordan canonical form, *SIAM Rev.*, *18*(4), 578–619.

Gorez, R. (1986a). The ICARE project—an interactive computing aid for research and engineering, personal communication, Laboratoire d'Automatique, de Dynamique et d'Analyse des Systèmes, Université Catholique de Louvain, Bâtiment Maxwell, Louvain-la-Neuve, Belgium.

Gorez, R. (1986b). PAAS—programme d'aide à l'analyse des systèmes. Personal Communication, Laboratoire d'Automatique, de Dynamique et d'Analyse des Systèmes, Université Catholique de Louvain, Bâtiment Maxwell, Louvain-la-Neuve, Belgium.

Gray, J. O. (1986). SANCAD and SATRES, personal communication, Dept. of Electronic and Electrical Engineering, University of Salford, Salford, United Kingdom.

Gupta, N. K., Groshans, D., and Houtchens, S. P. (1993). MATRIX_x, in: *CAD for Control Systems*, D. A. Linkens, ed., Marcel Dekker, New York, pp. 395–421.

Hanselman, D. C., and Kuo, B. C. (1995). *MATLAB Tools for Control System Analysis and Design*, 2nd Edition, Prentice Hall, Englewood Cliffs, NJ.

IBM (1984). *Dynamic Simulation Language/VS (DSL/VS). Language Reference Manual*, Program Number 5798:PXJ, Form SH20-6288-0, IBM Corp., Cottle Road, San Jose, CA.

Integrated Systems, Inc. (1984). *Matrix_x User's Guide, Matrix_x Reference Guide, Matrix_x Training Guide, Command Summary, and On-line Help*, Integrated Systems, Inc., Palo Alto, CA.

Integrated Systems, Inc. (1987). *SystemBuild User's Guide*, Santa Clara, CA.

Jamshidi, M., and Herget, C. J., (1993). *Recent Advances in Computer-Aided Control Systems Engineering*, Elsevier, Amsterdam.

Jamshidi, M. Tarokh, M., and Shafai, B. (1992). *Computer-Aided Analysis and Design of Linear Control Systems*, Prentice-Hall, Englewood Cliffs, NJ.

Jamshidi, M., Marchbanks, R., Bisset, K., Kelsey, R., Baugh, S., and Barak, D. (1993). Computer-aided design of fuzzy control systems: Software and hardware implementations, in: *Recent Advances in Computer-Aided Control Systems Engineering*, M. Jamshidi and C. J. Herget, eds., Elsevier, Amsterdam, pp. 81–126.

Kailath, T. (1980). *Linear Systems*, Prentice-Hall, Englewood Cliffs, NJ.

Kandel, A., and Langholz, G. (1994). *Fuzzy Control Systems*, CRC Press, Boca Raton, FL.

Karakasoglu, A. (1991). Neural network-based approaches to controller design for robot manipulators, Ph.D. Dissertation, University of Arizona, Tucson, AZ.

Klir, G. J. (1985). *Architecture of Systems Problem Solving*, Plenum Press, New York.

Koga, M., and Furuta, K. (1993). Programming language MaTX for scientific and engineering computations, in: *CAD for Control Systems*, D. A. Linkens, ed., Marcel Dekker, New York, pp. 287–317.

Korn, G. A. (1985). A new interactive environment for computer-aided experiments, *Simulation*, *45*(6), 303–305.

Korn, G. A. (1987). Control-system simulation on small personal-computer workstations, *Int. J. Modeling Simulation*, *8*(4).

Korn, G. A. (1989). *Interactive Dynamic-System Simulation*, McGraw-Hill, New York.

Korn, G. A., and Wait, J. V. (1978). *Digital Continuous System Simulation*, Prentice-Hall, Englewood Cliffs, NJ.

Laub, A. (1980). Computation of balancing transformations, *Proc. JACC, 1*, Paper FA8-E.

Lawver, B. (1985). EAGLES, an interactive environment and program development tool, personal communication, Dynamics and Controls Group, Lawrence Livermore National Laboratory, University of California, Livermore, CA.

Linkens, D. A. (1993). *CAD for Control Systems*, Marcel Dekker, New York.

Little, J. N. (1985). *PC-MATLAB,User's Guide, Reference Guide, and On-Line HELP, BROWSE, and Demonstrations*. MathWorks, Inc., Sherborn, MA.

Little, J. N., Emami-Naemi, A., and Bangert, S. N. (1984). CTRL-C and matrix environments for the computer-aided design of control systems, in: *Proceedings of the 6th International Conference on Analysis and Optimization (INRIA)*, Nice, France, Lecture Notes in Control and Information Sciences, Vol. 63(2), Springer-Verlag, New York, pp. 191–205.

Little, J. N., Herskovitz, S., Laub, A. J., and Moler, C. B. (1986). MATLAB and control design on the MacIntosh, in: *Proceedings of the 3rd Symposium on Computer-Aided Control Systems Design*, Washington, DC.

Maciejowski, J. M. (1984). Data structures for control system design, in: *Proceedings of the 6th European Conference of Electrotechnics, Computers in Communication and Control (EUROCON '84)*, Brighton, UK, Peter Peregrinus Ltd., London.

Maciejowski, J. M., and Szymkat, M. (1994). Containers—a step towards objects with MATLAB, in: *Proceedings of CACSD '94, Joint IEEE/IFAC Symposium on Computer-Aided Control System Design*, S. E. Mattsson, J. O. Gray, and F. E. Cellier, eds., Tucson, AZ, March 7–9, IEEE, Piscataway, NJ, pp. 277–285.

Maciejowski, J. M., and Taylor, J. H. (1994). A report on the 1993 IFAC World Congress Standards and Guidelines Session, in: *Proceedings of CACSD '94, Joint IEEE/IFAC Symposium on Computer-Aided Control System Design*, S. E. Mattsson, J. O. Gray, and F. E. Cellier, eds., Tucson, AZ, March 7–9, IEEE, Piscataway, NJ, pp. 377–380.

MathWorks, Inc. (1991). *SIMULINK User's Guide*, South Natick, MA.

MathWorks, Inc. (1992). *The Student Edition of MATLAB for MS-DOS or Macintosh Computers*, Prentice-Hall, Englewood Cliffs, NJ.

Mattsson, S. E., Andersson, M., and Åström, K. J. (1993). Object-oriented modeling and simulation, in: *CAD for Control Systems*, D. A. Linkens, ed., Marcel Dekker, New York, pp. 31–69.

Mattsson, S. E., Gray, J. O., and Cellier, F. E. (1994). *Proceedings of CACSD '94, Joint IEEE/IFAC Symposium on Computer-Aided Control System Design*, Tucson, AZ, IEEE, Piscataway, NJ.

Mitchell, E. E. L., and Gauthier, J. S. (1991). *ACSL: Advanced Continuous Simulation Language—User/Guide Reference Manual*, Mitchell & Gauthier, Assoc., Concord, MA.

Moler, C. (1980). *MATLAB User's Guide*, Dept. of Computer Science, University of New Mexico, Albuquerque, NM.

Monopoli, R. V. (1974). Model reference adaptive control with an augmented error signal, *IEEE Trans. Automatic Control, AC-19*, 474–484.

Mugica, F., and Cellier, F. E. (1994). Automated synthesis of a fuzzy controller for cargo ship steering by means of qualitative simulation, *Proceedings SCS European Simulation Multi-Conference*, Barcelona, Spain, A. Guasch and R. Huber, eds., SCS Publ., La Jolla, CA, pp. 523–528.

Munro, N., Palaskas, Z., and Frederick, D. K. (1986). An adaptive CACSD dialogue facility, in: *Proceedings of the 3rd Symposium on Computer-Aided Control System Design*, Washington, D.C.

Narendra, K. S. (1980). Recent developments in adaptive control, in: *Methods and Applications in Adaptive Control*, H. Unbehauen, ed., Springer-Verlag, New York.

Narendra, K. S., and Parthasarathy, K. (1990). Identification and control of dynamical systems using neural networks, *IEEE Trans. Neural Networks, 1*(1), 4–27.

Norsworthy, R., Kohn, W., and Arellano, J. (1985). A symbolic package for analysis and design of digital controllers, Honeywell, Inc., and NASA Johnson Space Center, private communication.

Ogata, K., (1994a). *Designing Linear Control Systems with MATLAB*, Prentice-Hall, Englewood Cliffs, NJ.

Ogata, K. (1994b). *Solving Control Engineering Problems with MATLAB*, Prentice Hall, Englewood Cliffs, NJ.

Otter, M. (1992). DSblock: A neutral description of dynamic systems, *OPEN-CACSD Electronic Newsletter, 1*(3), February 28.

Otter, M. (1995). *Dymosim—Dynamic Model Simulator*, Dynasim AB, Lund, Sweden.

Patel, R. V., and Misra, P. (1984). Numerical algorithms for eigenvalue assignment by state feedback, *Proc. IEEE, 72*(12), 1755–1764.

Pedrycz, W. (1989). *Fuzzy Control and Fuzzy Systems*, John Wiley & Sons, New York.

Rand Corp. (1985). *REDUCE User's Manual.* Santa Monica, CA.

Rimvall, M. (1983). *IMPACT, Interactive Mathematical Program for Automatic Control Theory, User's Guide*, Dept. of Automatic Control, Swiss Federal Institute of Technology, ETH-Zentrum, Zürich, Switzerland.

Rimvall, M., and Bomholt, L. (1985). A flexible man-machine interface for CACSD applications, in: *Proceedings of the 3rd IFAC Symposium on Computer-Aided Design in Control and Engineering Systems (CADCE '85)*. Copenhagen, July 31 to August 2, 1985, P. M. Larsen and N. E. Hansen, eds., Pergamon Press, Oxford, pp. 98–103.

Rimvall, M., and Cellier, F. E. (1985). The matrix environment as enhancement to modeling and simulation, in: *Proceedings of the 11th IMACS World Conference*, Oslo, August 5–9, 1985, North Holland, Amsterdam.

Rimvall, M., Frederick, D. K., Herget, C., and Kook, R. (1985). *ELCS—Extended List of Control Software, Newsletter*, Dept. of Automatic Control, ETH-Zentrum, Zürich, Switzerland.

Rosenbrock, H. H. (1969). Design of multivariable control systems using the inverse Nyquist array, *Proc. IEE, 116*, 1929–1936.

Sawyer, W. (1986). Polynomial operations with a trajectory representation, term project, M. Rimvall, advisor, Dept. of Automatic Control, ETH-Zentrum, Zürich, Switzerland.

Schmid, C. (1979). *KEDDC, User's Manual and Programmer's Manual*, Lehrstuhl für Elektrische Steuerung und Regelung, Ruhr University Bochum, Germany.

Schmid, C. (1985). KEDDC—a computer-aided analysis and design package for control systems, in: *Computer-Aided Control Systems Engineering*, M. Jamshidi and C. J. Herget, eds., North Holland, Amsterdam, pp. 159–180.

Shah, S., Shah, S. C., Floyd, M. A., and Lehman, L. L. (1985). Matrix$_x$: Control design and model building CAE capability, in: *Computer-Aided Control Systems Engineering*, M. Jamshidi, and C. J. Herget, eds., North Holland, Amsterdam, pp. 181–207.

Siljak, D. D., and Sundareshan, M. K. (1976). A multilevel optimization of large-scale dynamic systems, *IEEE Trans. Automatic Control, AC-21*, 70–84.

Spang, H. A., III (1984). The federated computer-aided control design system, *Proc. IEEE, 72*(12), 1724–1731.

Strandridge, C. R., and Pritsker, A. A. B. (1987). *TESS—The Extended Simulation Support System*, Halsted Press, John Wiley and Sons, New York.

Symbolics, Inc. (1983). *MACSYMA Reference Manual*, Version 10, MIT and Symbolics, Inc., Cambridge, MA.

Systems Control Technology (1984). *CTRL-C, a Language for the Computer-Aided Design of Multivariable Control Systems, User's Guide*, Systems Control Technology, Palo Alto, CA.

Systems Control Technology, Inc. (1990). *Model-C User's Guide*, Palo Alto, CA.

Systems Modeling Corp. (1985). *CINEMA User's Guide*, Systems Modeling Corp., State College, PA.

Taylor, J. H., and Frederick, D. K. (1984). An expert system architecture for computer-aided control engineering, *Proc. IEEE, 72*(12), 1795–1805.

Technical Software Systems (1985). *SSPACK User's Manual Including Sample Problems*, Technical Software Systems, Livermore, CA.

Thompson, P. M. (1986). Program CC, Version 3, personal communication, Systems Technology, Inc., Hawthorne, CA.

Tsao, L.-P. (1986). Interactive Nonlinear Programming, MS thesis, University of Arizona, Tucson, AZ.

Uyttenhove, H. J. (1979). *SAPS—System Approach Problem Solver*, Computing and Systems Consultants, Inc., Binghampton, NY.

Vanbegin, M., and Van Dooren, P. (1985). MATLAB-SC, Appendix B: Numerical Subroutines for Systems and Control Problems, Technical Note N168, Philips Research Laboratories, Bosvoorde, Belgium.

Van den Bosch, P. P. J. (1985). Interactive computer-aided control system analysis and design, in: *Computer-Aided Control System Engineering*, M. Jamshidi and C. J. Herget, eds., North Holland, Amsterdam, pp. 229–242.

West, P. J., Bingulac, S. P., and Perkins, W. R. (1985). L-A-S: A computer-aided control system design language in: *Computer-Aided Control Systems Engineering*, M. Jamshidi and C. J. Herget, eds., North Holland, Amsterdam, pp. 243–261.

Wieslander, J. (1980a). IDPAC Commands—User's Guide, Report: CODEN: LUTFD2/(TFRT-3157), Dept of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Wieslander, J. (1980b). MODPAC Commands—User's Guide, Report: CODEN: LUTFD2(TFRT-3158), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Wieslander, J. and Elmqvist, H. (1978). INTRAC, A Communication Module for Interactive Programs, Language Manual, Report: CODEN: LUTFD2/(TFRT-3149), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Wolovich, W. A. (1974). *Linear Multivariable Systems*, Springer-Verlag, New York.

Wonham, W. M. (1974). *Linear Multivariable Systems: A Geometric Approach*, Springer-Verlag, New York.