Appendix A

Computer-Aided Control Systems Techniques and Tools

A.1 INTRODUCTION

Up 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 as compared to Chapters 1 and 2, where the term "model representation" was used to denote graphical 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 independent 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 of 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 beside 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*, e.g., 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 in order 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 have tolerance values associated with them. It is to be determined 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 that have tolerances associated with them to be the parameters of the model. Set all parameters to their minimal values.
2. Perform a simulation.
3. Repeat 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 graphical 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 runs, and many other things in between. 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, very little was done with respect to enhancing the capabilities of simulation experiment descriptions (Cellier, 1986a). Some simulation languages, such as ACSL (Mitchell and Gauthier, 1981), 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, e.g., a means to compute the Fourier spectrum of a simulation trajectory. To our knowledge, there is not a single simulation system currently available that would offer a general-purpose nonlinear programming package for curve fitting, steady-state finding, the solution of boundary value problems, etc., as an integral part of the software, and this is only one among many tools that would be useful to have. Moreover, the few experiment enhancement tools that are currently available are often difficult to use and are very specialized, that is limited in applicability.

Whenever such a situation is faced, we, as software engineers, realize that something must 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 1967, 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, but simply one software module (tool) among many others that can be called at will from within the "experiment description software," which from now on will be called "computer-aided design software." Algorithms for particular purposes will be called *computer-aided design techniques,* and the programs implementing these algorithms will be called *computer-aided design tools*. As many of the design tools are application dependent, our discussion will be restricted to one particular application, namely, the design of control systems.

Until very recently, 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 pitiable. Users were led through an inflexible question-and-answer protocol. Once an incorrect specification was entered by mistake, there was no 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 was forced 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]
```

or alternatively:

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

Elements in different columns are separated by either comma or space, whereas elements in different rows are separated by either semicolon or carriage return (CR). With matrices being 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]]
```

where ONES(3,1) stands for 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 which is concatenated to the 0 matrix from the right, thus making the total 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 above expression will create the matrix

$$A = [\begin{matrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -2 & -3 & -4 & -5 \end{matrix}]$$

Suppose it is desired to solve the linear system

$$\underline{A}*\underline{x} = \underline{b}$$

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

$$\underline{x} = \underline{A}^{-1}*\underline{b}$$

which, in MATLAB, can be expressed as

$$\underline{x} = \mathrm{INV}(\underline{a})*\underline{b}$$

or, somewhat more efficiently,

$$\underline{x} = \underline{a}\backslash\underline{b}$$

($\underline{b}$ from left divided by $\underline{a}$), in which case a Gaussian elimination is performed in place of the computation of the complete inverse. With MATLAB, we finally got a tool that allows us to learn what we always wanted to know about linear algebra [such as: what are the EIG($\underline{a}$+2$\underline{\text{EYE}}$($\underline{a}$)) where $\underline{\text{EYE}}$($\underline{a}$) stands for a unity matrix with the same dimensions as $\underline{a}$, and EIG(...) computes the eigenvalues of the enclosed expression]. In fact, such a tool existed already 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. Cleve Moler, on the other hand, taught the computer to "think" like the human operator.

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 that the control engineer needs for solving his problems. As an example, let us solve a simple LQG (linear quadratic gaussian) regulator design problem. For the linear system

$$\frac{d\underline{x}}{dt} = \underline{A}*\underline{x} + \underline{B}*\underline{u}$$

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

$$PI = \int_0^\infty (\underline{x}'\underline{Q}\underline{x} + \underline{u}'\underline{R}\underline{u})dt \overset{!}{=} \min$$

where $\underline{u}'$ denotes the transpose of the vector $\underline{u}$. 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:

$$\underline{H} = [\begin{matrix} \underline{A} & -\underline{B}\underline{R}^{-1}\underline{B}' \\ -\underline{Q} & -\underline{A}' \end{matrix}]$$

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

$$\underline{V} = [\begin{matrix} \underline{V}_1 \\ \underline{V}_2 \end{matrix}]$$

5. Now, the Riccati feedback can be computed as

$$\underline{K} = -\underline{R}^{-1}\underline{B}'\underline{P}$$

where

$$\underline{P} = \mathrm{Re}\{\underline{V}_2 * \underline{V}_1^{-1}\}$$

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 had become available, it took amazingly little time until several CACSD experts realized that this was an excellent way to express control problems. Clearly, MATLAB was not designed for CACSD problems, and still a lot 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, 1984; Little et al., 1984), MATRIX$_X$ (Integrated Systems, 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). All these programs are "spiritual children" of MATLAB.

We want to demonstrate in this appendix how simulation software designers can learn from recent experiences in CACSD program development, and how the CACSD program developers can learn from the 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. It would, indeed, be useful if the simulation software could apply to linear (sub)systems an integration algorithm that is more efficient than the regularly used explicit Runge Kutta, Adams-Bashforth, or Gear algorithms, e.g., an implicit Adams-Moulton technique. Linear (sub)systems could automatically be identified by the compiler through the use of a matrix notation.

On the other hand, it is true that most recent CACSD programs offer only limited simulation capabilities (e.g., applicable to linear systems only). It would be useful indeed if all our knowledge about the simulation of dynamic systems/processes could be integrated into the CACSD software. Since a design usually involves more than just simulation, but certainly simulation among other techniques, it would definitely be beneficial if a flexible interface between a CACSD program and a simulation language could be created such that powerful simulation runs could be made efficiently at arbitrary points in a more complex design study. These are some of the topics that this appendix addresses.

Note that this appendix discusses purely digital solutions only. Other simulation techniques (such as analog and/or hybrid simulation techniques) are discussed in Chapter 4 of this book. 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) will then be programmed on the analog part of the computer, while the experiment description that triggers indivual simulation runs will be programmed on the digital part of the computer. The digital CACSD program will look just the same as in the purely digital solution, while the simulation program will look just 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 Section A.3, CACSD tools are classified. This discussion highlights the major differences between several classes of CACSD tools.

Both Sections A.2 and A.3 help to prepare for Section A.4, in which a number of currently available CACSD tools are compared with respect to features (algorithms) offered by these software systems.

After this discussion, the reader may question whether a diversification of tools as can be currently observed is truly justified and desirable. For this reason, the problem of software standardization versus software diversification is discussed in Section A.5.

In Section A.6, we show how simulations can intelligently be used within CACSD software. This section helps to throw a bridge across to other chapters in this book.

Finally, Section A.7 presents our perspective of forthcoming developments in the area of CACSD software design.

A.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, weapon 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.)

In the beginning, that is, in the 1930s–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 people such as W. R. Evans and H. Nyquist. Most of the techniques developed were graphical in nature.

With the need to deal with more complex systems with multiple inputs and outputs (so-called MIMO systems), these graphical techniques failed to provide sufficient insight. It was among others R. 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 nth-order ODEs. For further detail, refer to Chapter 2 of this book and to the References of Chapter 2. 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, as frequency domain design can be done manually also in the case of higher-order systems.) What is described above is the technology of the sixties.

What has happened since? What were the major breakthroughs in the seventies and in the early eighties? While research in control theory before was pretty much consolidated, now diversification took place; that is, 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, called on a subroutine, and the answer came in the form of some other numbers, parameter values, gain factors, etc. 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 do we go from there? Often, the conclusion had been 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 is going on. None of the automated algorithms available at that time was 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. For a somewhat cheaper solution, it may be possible to employ sensitivity analysis instead (Cellier, 1986a). The newest developments in this area try to do away with numerical algorithms altogether. Instead of computing numerically one point in the parameter space at a time, the new algorithms reproduce what the engineer used to do in the paper and pencil age, that is, formula manipulation. The latest developments in nonnumerical data processing are employed to obtain algorithmically and automatically a formula that relates the desired output parameters to the given input parameters. However, these techniques are still in their infancy, and no commercial product of this kind is available as of today. The furthest developed program of this type known to us is an extensive LISP program running on an LMI computer (a special-purpose LISP machine). That program is currently under development at NASA's Johnson Space Center, but it is far from completed, and its user interface is still rather awkward (Norsworthy et al., 1985).

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, or 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 cut the system into smaller subsystems and find answers for those subsystems first. This led to decentralized control (Athens, 1978) and hierarchical control (Siljak and Sundareshan, 1976) schemes. More information about these approaches can be found in Chapter 6 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 use of a digital computer. This stimulated research into discrete-time algorithms, as 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 for developing 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 at 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 is, the less likely will it be to handle modified situations. 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 *self-tuning regulators* (Åström, 1980), *model-reference adaptive controllers* (Monopoli, 1974; Narendra, 1980), and *robust controllers* (Ackermann, 1980).

So far, we have presented the problems to be solved. Problems can be classified into single-input, single-output (SISO), multiple-input, multiple-output (MIMO), 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 numerical aptness of an algorithm, of numerical accuracy and numerical stability. The numerical behavior of algorithms is highly dependent on the system order, that is, the number of state variables describing the system/process. Almost any algorithm can be used to solve a third-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, and this is true for almost every single algorithm in all three classes of problem types. This fact was detected only recently. Since the late seventies, many researchers, including C. Moler, G. H. Golub, A. Laub, J. H. Wilkinson, and P. 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 (SVD), described in Golub and Wilkinson (1976).

From now on, algorithms that work only for low-order systems will be referred to as LO algorithms, techniques that work also for high-order systems will be called HO algorithms, and finally, methods that can be used to treat very-high-order systems (mostly discretized distributed parameter systems) will be 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 canonical forms (Kailath, 1980). All these canonical 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

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

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; i.e., 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-canonical representation of this system can be written as

$$\dot{\underline{x}} = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ -a_0 & -a_1 & -a_2 & -a_3 & \dots & -a_{n-1} \end{bmatrix} \underline{x} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \dots \\ 1 \end{bmatrix} u$$

$$y = [\ b_0 \quad b_1 \quad b_2 \quad b_3 \quad \dots \quad b_{n-1}\ ]$$

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 above. Also, the Jordan-canonical representation

$$\dot{\underline{x}} = \begin{bmatrix} \lambda_1 & 0 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & 0 & \dots & 0 \\ 0 & 0 & \lambda_3 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & \lambda_n \end{bmatrix} \underline{x} + \begin{bmatrix} 1 \\ 1 \\ 1 \\ \dots \\ 1 \end{bmatrix} u$$

$$\underline{y} = [r_1 \quad r_2 \quad r_3 \quad r_4 \quad \dots \quad r_n]\underline{x}$$

(assuming all eigenvalues λ_i to be distinct) has exactly the same number of parameters. This is true for all the canonical forms. For LO systems, these representations are perfectly acceptable. However, we require redundancy in order to optimize the numerical behavior of algorithms for HO systems. Thus, all algorithms that are based on canonical 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 numerical 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, 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* have a certain overhead associated with them. 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 above-described algorithms for HO problems, sparse matrix techniques do not influence the numerical behavior of the involved algorithms, but only their execution time. Therefore, the numerical problems discussed for the case of 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 classical canonical 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 with each other. 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. It is our opinion 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 numerical sensitivity of an algorithm, one has to balance the sensitivities of the system parameters; i.e., 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 sensitivities 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 will lead to extremely unbalanced parameter sensitivities. Let us consider the polynomial

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

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, while all other parameters are not. However, if we evaluate the polynomial at $s = 1000$, obviously a_n exerts the strongest influence, while a_0 can easily be neglected. This problem disappears when we represent the polynomial through its roots:

$$q(s) = k(s - s_1)(s - s_2) \cdot \cdot \cdot (s - s_n)$$

Here, the parameters are (k) and $(s_1 \ . \ . \ . \ s_n)$ instead of $(a_0 . \ . \ . \ a_n)$. However, if we want to add two polynomials, we won't get around to (at least partially) defactorize the polynomials and refactorize 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 numerical algorithms based on these data representations as 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 polynimial by regression analysis, reducing the numerical errors involved in this computation. The basic operations (addition, subtraction, multiplication, division) all become trivial in this data representation if all involved polynomials are evaluated over the same base of supporting values (we just apply them to each data point separately), and as 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 we feel that this approach could possibly lead to a breakthrough in the numerical algorithms for frequency domain operations.

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 numerical and nonnumerical algorithms. Nonnumerical algorithms make use of formula manipulations, and "reasoning" techniques that are usually connoted as artificial intelligence (AI) techniques. Since none of the CACSD programs discussed in Section A.4 makes extensive use of such techniques, we shall save a more intimate discussion of AI techniques for the "outlook" section (A.7).

A.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 (CACSD) really did not start until 1970. At that time, it would take roughly half a day just to find the eigenvalues of a matrix, as this involved the following procedure:

1. Develop a program to calculate eigenvalues by calling a library subroutine with about six call parameters (1/2 hr).
2. Walk to the computer center to prepare the data input (20 min).
3. Wait for a card puncher to become available (1/2 hr).
4. Prepare input data (10 min).

5. Submit card deck to input queue, and wait for output to be returned (turnaround time roughly 1 hr).
6. Correct typos after waiting for another card puncher to become available, and resubmit card deck; wait again for output (90 min).
7. Walk back to office (20 min).

The solution of a true control problem (e.g., the simple LQG design problem described above) took 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 on other topics that did not require any involvement in this denervating process.

Around 1972, the writers undertook the effort to go around and ask colleagues in the department not to throw away their control programs any longer (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, as 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 on 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 to many became real fun. However, as a research tool, INTOPS failed to provide the necessary flexibility, and we became convinced that a true full-fledged programming language was required for that purpose. Unfortunately, such a language could no longer be considered as a small and inexpensive add-on to the control library, and we wondered what could be done about it.

In the fall of 1980, K. J. Åström and G. Golub undertook the commendable effort to bring recognized numerical analysts and control experts together in the first Conference on Numerical Techniques in Control to be ever held. On this occasion, we met with Cleve 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 belongs to the department of electrical engineering). Students were able to learn the usage of this tool within half an hour, and suddenly, also the researchers 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 to use. Who can remember all those commands and their parameters except someone who uses the tool on a daily basis? It 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.

Notice that MATLAB really was not designed to be a CACSD tool at all. There are many shortcomings of 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 available; and the input/output capabilities of EXEC-files are too limited.
2. The SAVE/LOAD concept of MATLAB is insufficient as 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, the user wants the possibility to interface data produced/used by his own programs with MATLAB.
3. Control engineers prefer results in graphical form. The output facilities offered by MATLAB are insufficient in every respect. A database interface would at least soften this request, as 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, adaptive controllers for time-varying systems, etc.). MATLAB does not provide for 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.)

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

To summarize this section: CACSD tools can be classified into subprogram libraries versus integrated design suites. The first generation of CACSD tools was of the former type; the more recent ones are mostly of the latter type. This new type of CACSD tools 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 own algorithms within the frame of the CACSD software; MATLAB (Mohler, 1980) is an example of that 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. Those 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 use of 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 which are 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. Compiler-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 either question-and-answer-driven, command-driven, menu-driven, form-driven, graphics-driven, or window-driven. The user of the program is asked questions to determine what needs to be computed next. Thus, the program flow is completely predetermined. 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 just the same manner as an interactive operating system (e.g., on a PDP 11 or on a VAX) 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 just sending a prompt. It then waits for the user to pick one of the items on the list, normally by use of a pointing device such as a cross-hair cursor or a Swiss mouse. This interface type is quite easy to implement, and it can be very powerful. One of its major drawbacks lies in the amount of information that must be exchanged between the program and the user. This interface type is simply too slow; moreover, pointing devices have not yet been standardized, making the software terminal-dependent.

A form-driven interface is most profitably used during the setup period of the CACSD 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 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 new standard. A laudable exception is DEC, which adopted the new ANSI standard when switching from the VT52 series terminals to the VT100 series terminals. Owing to these hardware dependency problems, most portable CACSD programs do not exploit this interface type either.

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 graphical 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 dependency 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 have been developed (DISSPLA, DI-3000, etc.). Unfortunately, all these commercially available libraries were expensive, and no true standard existed. Recently, the graphics kernel system (GKS) has been accepted as a standard (ANSI, 1985), and this will certainly drive the prices for GKS implementations down. Fancy graphics, however, 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.

A first generation of simulation programs was recently made available that offer 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 (either slower, equal, 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 (Pritsker & Assoc., 1986) and CINEMA (Systems Modelling 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).

Graphical input has also become a reality. Control circuits can be drawn on the screen by means of block diagrams, which then are automatically translated by a graphical compiler into a coded model representation. MATRIX$_X$ (Integrated Systems, 1984, Shah et al., 1985) already provides this feature when operated from a SUN workstation (the program module implementing this feature is called SYSTEM-BUILD). The most fancy implementation, however, is provided in HIBLIZ (Elmqvist, 1982; Elmqvist and Mattson, 1986). This program uses a virtual screen concept similar to the one 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 hierarchical. 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 where 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 either 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.

A window-driven interface permits splitting the physical screen into several logical windows. Each window is now associated with one logical unit in just the same manner as different physical devices used to be. Each window by itself can theoretically be alphanumerical or graphical, 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 that case, the most recently addressed window will automatically become the top window that is totally visible. On some occasions (e.g., the AMIGA operating system), 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 by 1000 pixels).

The different interface modes described above are by no means incompatible with each other. We recently experimented with combinations of several interface types. IMPACT (Rimvall, 1983; Rimvall and Bomholt, 1985; Rimvall and Cellier, 1985) 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, e.g., 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 use of a SPAWN facility that works similarly to the one 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 graphical operating system where the windows themselves are sometimes menu-driven and sometimes form-driven.

A.4 CACSD TOOLS—A SURVEY

As discussed above, a CACSD program can offer a variety of interfaces. There are also different types of design problems to be solved, and a CACSD program may be either general or specialized. Thus, it is important to select the right tool for the problem to be solved. The available CACSD programs are by no means uniform.

Therefore, we gathered information about some of the currently available CACSD programs. We developed a list of features using the categories described above and mailed it to the producers of all CACSD programs we were aware of. There were at that time roughly 40, and half of them responded. Their answers are listed in Table A.1. On the left, the features are listed; thus each feature occupies one row. On the top, the different CACSD programs are shown; thus each program occupies one column. A 0 entry denotes the fact that the feature is not available in the particular CACSD program, a 1 entry denotes availability, and a 2 entry denotes particular strength. In the last row of the table, the accumulated "score" of each CACSD program is presented.

In Table A.1, whenever there existed several versions of one program, we listed the features of the strongest version. For example, MATRIX_X, (Integrated Systems, 1984; Shah et al., 1985) has considerably stronger power when executed from a SUN workstation than when operated through a VT100 on a VAX system. The features listed are consequently those of the workstation implementation. The mainframe version has substantially fewer accumulated points and is roughly comparable to CTRL-C or IMPACT (two other mainframe programs). As another example, CTRL-C (Systems Control, 1984; Little et al., 1984) offers only a few features for the treatment of nonlinear systems by itself, but there exists a strong link between CTRL-C and the simulation language ACSL (Mitchell and Gauthier, 1981), making it possible to run ACSL simulations under the supervision of CTRL-C. Thus, CTRL-C is used for the experimental design, while ACSL is utilized for individual simulation runs. The features listed under the heading "CTRL-C" are consequently those of the combined software system CTRL-C/ACSL.

Columns 2–7 list features of MATLAB and its "spiritual children." These programs are actually all very similar in nature and therefore easy to compare. However, the difference in power between the various programs is dramatic. As can be seen from Table A.1, the original MATLAB (Moler, 1980) accumulated 39 points only (and is thus the second "weakest" CACSD program listed), whereas the "strongest" of these programs, MATRIX_X (Integrated Systems, 1984; Shah et al., 1985), accumulated 149 points (more than any other of the listed programs). Also CTRL-C (Systems Control, 1984; Little et al., 1984) and IMPACT (Rimvall, 1983; Rimvall and Bomholt, 1985; Rimvall and Cellier, 1985) are very strong programs, and even the "small" PC-MATLAB (Little, 1985) is amazingly powerful. The developers of MATLAB-SC (Vanbegin and Van Dooren, 1985) concentrated on the algorithmic aspects, implementing some powerful, numerically stable algorithms for particular control problems, but this is only one of the six previously mentioned shortcomings of MATLAB. Therefore, MATLAB-SC accumulated relatively few additional points over MATLAB itself and cannot be considered a full-fledged CACSD tool yet. All the other programs (in Columns 2–7) added strength in face of the six deficiencies in one form or the other.

MATRIX_X, CTRL-C, and MATLAB-SC are true descendants of MATLAB, in that they started off from the (FORTRAN-coded) MATLAB program and added enhancements where they saw fit.

PC-MATLAB was completely recoded in C, making its code somewhat easier to read and maintain. PC-MATLAB was developed for operation on the IBM-PC and its lookalikes, which actually was one of the reasons for recoding PC-MATLAB from scratch (the C-compiler available for the PC is far better than the FORTRAN-compiler). Since the time of our questionnaire, two additional implementations of PC-MATLAB were made available, namely, one for the MacIntosh and the other for the SUN workstation, and therefore, the name of the software was changed from PC-MATLAB to PRO-MATLAB (Little et al., 1986). Also since the time of our questionnaire, a tight link between PRO-MATLAB and the simulation language SIMNON (Elmqvist, 1975, 1977) was established. With this new link, PRO-MATLAB passes the 100-point level, making the software comparable in power to CTRL-C and IMPACT.

Also, IMPACT was completely recoded using ADA. IMPACT's particular strength lies in its powerful data structures, which will be discussed later.

One more derivative of MATLAB, called M (Gavl and Herget, 1984), has also been completely recoded using OBJECTIVE-C. This implementation is very powerful. M may be of particular interest to universities, as it is the only one of MATLAB's derivatives that is available as public domain software (as the original MATLAB was). However, its developers do not want to advertise M too highly before its implementation has come to an end, and thus, they did not respond to our questionnaire. Therefore, M's features are regrettably not contained in Table A.1.

Before we continue with the analysis of other programs listed in Table A.1, let us discuss some of the concepts that went into the design of MATLAB's derivatives. Why, for instance, did we decide to reimplement IMPACT from scratch rather than making use of the code already available in MATLAB? It is our experience that new program structures can be easily added on to an existing code, while it is almost hopeless to try to add new data structures to it. Thus, one is stuck with a code once one decides that a major review of the available data structures is needed. As the complex double-precision matrix is the only available data structure in MATLAB, the "true" children of MATLAB had to squeeze any new data types into this data representation. For example, in CTRL-C, polynomials such as p(s)

$$p(s) = 5s^3 + 8s^3 - 2s + 7$$

are represented through their coefficients coded as a vector.

$$\underline{P} = [5 \quad 8 \quad -2 \quad 7]$$

Table A.1 CACSD Program Features

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1		MATLAB	CTRL-C	IMPACT	MATLAB-SC	MATRIX$_x$	PC-MATLAB	CATPAC	CC	KEDDC	L.A.S.	LUND	TRIP	WCDS/DSC	ICARE	MADPAC	PAAS	SANCAD	SSPACK	SUBOPT	SUNS
2																					
3																					
4																					
5																					
6																					
7	GLOBAL CLASSIFICATION																				
8																					
9	Continuous Systems	0	2	2	1	2	2	1	2	2	1	2	2	2	2	0	1	1	0	2	2
10	Discrete Systems	0	2	2	1	2	2	1	2	2	1	2	2	2	0	2	0	1	2	0	2
11	Time Domain	0	2	2	1	2	2	1	2	2	1	2	2	2	2	0	0	0	2	2	0
12	Frequency Domain	0	2	2	0	2	2	1	2	2	1	2	2	2	2	2	2	1	0	0	2
13	Single-Input/Single-Output	0	2	2	1	2	2	1	2	2	1	2	2	2	2	2	2	1	2	2	2
14	Multivariable	0	1	2	0	1	1	1	2	2	1	1	0	2	2	0	0	1	0	2	2
15	Nonlinear Systems	0	2	1	0	2	0	1	0	1	1	1	1	0	0	0	0	2	1	0	2
16	Adaptive Control	0	0	0	0	1	1	0	0	2	0	2	0	0	0	2	0	0	0	0	0
17	Identification	0	1	1	0	1	1	1	0	2	1	2	1	0	0	2	0	0	2	1	0
18	Real-Time Interface	0	0	0	0	2	0	0	0	2	0	2	0	0	0	0	0	0	0	0	0
19																					
20																					
21	USER INTERFACE																				
22																					
23	Interactive Operation																				
24	Input																				
25	Question-and-Answer	0	0	1	0	1	0	1	1	2	1	1	0	0	1	2	2	1	2	2	1
26	Menu-/Form-driven	0	0	2	0	2	0	0	0	1	0	0	0	0	1	1	2	1	2	0	1
27	Object oriented	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
28	Command-driven	2	2	2	2	2	2	1	2	1	0	2	2	2	0	0	0	0	1	0	0

29	Full interactive language	1	2	2	1	2	2	0	0	0	2	1	0	0	0	0	0	0	0	0	0
30	Graphical input (Model spec.)	0	0	0	0	2	0	0	0	0	0	1	0	0	0	0	0	2	0	0	0
31	Window—handler	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
32	Parallel sessions	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
33	Graphical Output																				
34	Print—plots	0	2	0	0	2	0	0	1	1	1	1	1	0	0	1	1	1	0	0	0
35	"Pen" plots	0	2	2	2	2	2	1	1	2	1	2	2	1	0	1	0	0	2	2	1
36	Frequency plots (Bode, etc.)	0	2	2	0	2	2	1	2	2	1	1	1	1	0	0	1	1	2	0	1
37	Time transients	0	2	2	0	2	2	1	2	2	1	2	1	1	0	1	1	1	2	2	0
38	General plots	0	2	1	1	1	2	0	0	2	0	1	0	1	0	0	0	0	1	0	1
39	3—dimensional plots	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
40	Split—screen graphics	0	2	2	0	2	0	0	0	2	0	2	2	0	0	0	0	1	2	0	0
41	Window handler	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
42	Color graphics	0	0	2	2	2	0	0	0	2	0	2	2	0	0	2	0	2	0	0	0
43																					
44	Batch operation																				
45	Callable from other program	1	0	0	1	0	0	1	1	0	1	1	0	0	0	0	0	0	1	0	0
46	Detached execution	0	1	1	0	1	0	1	0	0	1	0	0	1	0	0	0	0	1	0	0
47																					
48	Extendability																				
49	Interactive macro definition	1	2	2	2	2	2	0	2	0	1	1	0	1	0	0	0	0	0	0	0
50	Interactive subprograms	0	2	2	2	2	2	0	1	0	1	0	0	0	0	0	0	0	0	0	0
51	Macro/subprogram libraries	0	2	2	0	1	1	1	1	2	1	1	0	1	0	0	0	1	1	0	0
52	Program interface new algor.	1	2	1	1	1	0	1	1	1	1	1	0	0	1	1	1	1	1	0	1
53	Data—exchange over data—base	0	0	1	0	1	1	0	1	1	1	0	0	0	0	0	0	1	1	0	0
54																					
55	Documentation																				
56	User's Manual	2	2	2	2	2	2	2	2	2	1	1	1	1	0	1	0	1	2	2	1
57	Machine readable	2	0	2	2	0	0	2	0	1	1	1	0	0	0	0	0	0	2	0	0
58	Tutorial	0	1	1	1	1	2	1	1	1	0	1	1	0	1	1	1	1	2	2	0
59	On—line help	1	1	2	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1

Table A.1 (Continued)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
60		MATLAB	CTRL-C	IMPACT	MATLAB-SC	MATRIX$_x$	PC-MATLAB	CATPAC	CC	KEDDC	L.A.S.	LUND	TRIP	WCDS/DSC	ICARE	MADPAC	PAAS	SANCAD	SSPACK	SUBOPT	SUNS
61																					
62																					
63																					
64																					
65																					
66	DATA/PROGRAM STRUCTURES																				
67																					
68	Numerical Structures																				
69	Matrix environment	2	2	2	2	2	2	1	1	1	1	1	2	1	2	1	0	0	2	0	0
70	Polynomial matrices	0	0	2	0	1	0	0	1	1	1	1	0	2	2	0	0	1	0	0	0
71	Transfer–function structures	0	1	2	0	1	1	1	2	1	1	1	2	2	2	0	2	1	0	0	1
72	State–space representation	1	1	2	2	2	1	1	2	1	1	2	2	1	2	1	0	0	2	2	0
73	System topologies	0	1	1	0	2	1	0	1	1	0	1	0	1	0	0	0	1	0	0	0
74																					
75	Symbolic structures																				
76	General formula descriptions	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0
77																					
78	Nonnumeric modelling structures																				
79	Nonlinearities	0	1	1	0	2	0	1	0	0	1	1	0	0	0	0	0	2	1	0	1
80	Nonlinear system descriptions	0	2	1	0	2	0	1	0	1	1	1	2	0	0	0	0	2	1	0	2
81	Continuous	0	2	1	0	2	0	0	0	1	1	1	2	0	0	0	0	2	0	0	2
82	Discrete	0	1	1	0	1	0	0	0	1	1	1	0	0	0	0	0	2	1	0	2
83	Sampled data systems	0	2	1	0	2	0	1	0	1	1	1	0	0	0	0	0	2	1	0	2
84	Partial differential equations	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
85	Hierarchical model definition	0	1	1	0	2	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0
86																					
87	Command structures																				
88	Mathematical notation	2	2	2	2	2	2	0	1	0	1	1	0	1	0	0	0	0	0	0	0

89	Structural elements (FOR, IF ...)	1	2	2	1	2	2	0	0	0	1	1	0	0	0	0	0	0	0	0	0
90	Macro execution																				
91	Interpretation	1	2	2	1	2	2	0	1	0	1	2	0	1	0	0	0	0	0	0	0
92	Direct executing code	0	1	2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
93	Automatic compilation	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
94																					
95	User–defined structures																				
96	Clustering of predefined struct.	0	1	2	0	2	1	0	1	0	1	0	0	1	0	0	1	1	0	0	0
97	New numeric structures	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
98	New symbolic structures	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
99																					
100																					
101	BUILT–IN ALGORITHMS																				
102																					
103	Mathematical Routines																				
104	Basic linear algebra routines	2	2	2	2	2	2	2	2	2	1	2	1	1	1	1	1	0	1	2	2
105	Eigenvalue computations	2	2	2	2	2	2	2	2	2	2	2	1	1	1	0	0	0	0	2	0
106	Decompositions, transformat.	2	2	2	2	2	2	2	2	2	1	2	0	1	1	0	0	0	0	2	0
107	Polynomial operations	0	1	2	1	1	1	1	2	2	1	1	1	2	1	1	1	1	0	0	0
108	Transfer–function operations	0	1	1	0	1	1	1	2	2	1	1	1	1	0	0	2	1	0	0	0
109	Nonlinear equation solving	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
110																					
111	General data analysis																				
112	Random number generator	0	1	2	1	1	1	1	1	1	1	1	0	1	0	1	0	1	1	0	1
113	Statistical analysis	0	0	2	0	1	1	0	0	1	1	1	0	0	0	0	0	0	1	0	0
114	Fourier–transforms	0	1	0	0	1	1	0	0	1	1	1	1	0	0	0	0	0	1	0	0
115	Filter design	0	1	0	0	1	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0
116																					
117	Symbolic calculations																				
118	Formula manipulation	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
119	Parameterized studies	0	1	0	0	1	0	0	1	0	0	1	0	1	0	1	0	0	0	1	0

Table A.1 (Continued)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
120		MATLAB	CTRL–C	IMPACT	MATLAB–SC	MATRIX$_x$	PC–MATLAB	CATPAC	CC	KEDDC	L.A.S.	LUND	TRIP	WCDS/DSC	ICARE	MADPAC	PAAS	SANCAD	SSPACK	SUBOPT	SUNS
121																					
122																					
123																					
124																					
125																					
126	Transformation routines																				
127	Similarity transformations	2	2	2	2	2	2	0	2	2	1	1	0	1	1	0	0	0	0	0	0
128	State–space <–> frequency dom.	0	2	2	0	2	2	0	2	1	1	2	1	1	1	1	0	0	0	0	0
129	Continuous <–> discrete	0	2	2	0	2	2	1	2	2	1	2	1	1	0	0	0	0	1	0	0
130	Subsystem interconnections	0	1	1	0	2	1	1	1	0	1	1	0	1	0	0	0	0	0	0	0
131																					
132	Identification																				
133	Scope																				
134	Continuous systems	0	1	1	0	2	1	0	0	2	1	2	2	0	0	0	0	0	0	2	0
135	Sampled–data systems	0	1	0	0	1	1	1	0	2	1	2	2	0	0	1	0	0	2	0	0
136	Linear–in–parameter systems	0	0	1	0	1	1	1	0	2	1	2	2	0	0	1	0	0	2	1	0
137	General parameters	0	0	0	0	1	1	1	0	1	0	1	2	0	0	0	0	0	2	0	0
138	Methods																				
139	Maximum likelihood	0	1	1	0	2	0	1	0	2	0	2	0	0	0	0	0	0	0	0	0
140	Least–squares	0	1	1	0	1	1	1	0	1	1	2	2	0	0	2	0	0	2	0	0
141																					
142	Model reduction																				
143	Modal reduction	0	1	1	0	2	1	1	2	0	1	2	1	1	0	0	0	0	0	0	0
144	Singular perturbation	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
145																					

146	Analysis routines																				
147	Controllability, observability	0	2	2	2	2	2	0	2	1	1	2	0	1	1	0	0	0	0	0	0
148	Stability	0	2	2	2	2	2	1	2	1	1	2	1	1	1	0	1	0	0	1	2
149	Poles, zeroes	1	1	1	1	1	1	1	2	1	1	1	2	2	1	0	1	0	0	0	0
150	Frequency responses	0	2	2	0	2	2	0	2	1	1	2	2	1	0	0	1	1	0	0	1
151	Time responses	0	2	2	0	1	2	1	2	1	1	2	1	1	0	1	1	1	1	2	0
152	Continuous linear systems	0	2	2	0	2	2	0	2	1	1	2	1	1	0	0	1	1	0	2	0
153	Discrete systems	0	2	2	0	2	2	0	2	1	1	2	1	1	0	1	0	1	1	0	0
154	Sampled–data systems	0	2	0	0	2	1	1	2	1	1	2	1	0	0	0	0	1	1	0	0
155	Nonlinear systems	0	1	1	0	2	0	0	0	1	1	2	2	0	0	0	0	2	2	0	2
156	Simulation	0	2	1	0	2	0	0	0	1	1	2	2	0	0	0	0	2	2	0	0
157	Stability tests	0	2	0	0	1	0	0	0	1	0	0	0	0	0	0	0	2	0	0	2
158	Steady–state analysis	0	2	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
159	Sensitivity analysis	0	1	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
160	Linearization	0	1	1	0	1	0	0	0	0	0	1	2	0	0	0	0	1	0	0	0
161																					
162	Design routines																				
163	Classical controller design	0	1	1	1	1	1	0	2	1	1	1	2	1	1	0	0	1	0	0	1
164	Pole placement/Observer des.	0	2	2	2	2	2	0	2	1	1	2	0	0	1	1	0	0	0	0	0
165	LQG design/Kalman filters	0	2	2	2	2	2	0	2	1	1	2	1	2	1	2	0	0	1	2	0
166	Opt. control (e.g. time–o.c.)	0	0	0	0	1	1	1	0	1	1	0	0	0	0	0	0	0	0	2	0
167	Parameter optimization	0	0	1	0	1	0	1	0	1	1	2	0	1	0	0	0	0	0	2	0
168	Multivariable design																				
169	Time domain	0	1	1	0	1	1	0	1	1	1	1	0	1	0	0	0	0	0	2	0
170	Frequency domain	0	0	0	0	1	1	0	1	1	1	0	0	2	0	0	0	0	0	0	1
171	Adaptive control design	0	0	0	0	1	1	0	0	2	0	2	0	0	0	2	0	0	0	0	0
172	Nonlinear design	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	0	2
173	Distributed systems design	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table A.1 (Continue

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
174		MATLAB	CTRL-C	IMPACT	MATLAB-SC	MATRIX$_x$	PC-MATLAB	CATPAC	CC	KEDDC	L.A.S.	LUND	TRIP	WCDS/DSC	ICARE	MADPAC	PAAS	SANCAD	SSPACK	SUBOPT	SUNS
175																					
176																					
177																					
178																					
179																					
180	REAL TIME CAPABILITIES																				
181																					
182	Real–time synchronization	0	0	0	0	2	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
183	Data gathering	0	0	0	0	2	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
184	Real–time control	0	0	0	0	2	0	0	0	2	0	1	0	0	0	0	0	0	0	0	0
185	Code generation	0	0	0	0	2	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
186																					
187																					
188	PORTABILITY, RELIABILITY																				
189																					
190	General portability																				
191	Portable programming language	2	1	2	2	1	1	2	1	2	1	1	2	2	1	1	1	1	1	2	1
192	Portable hardware interface	2	1	1	2	1	1	1	1	1	0	0	1	0	0	0	0	1	0	2	0
193	Foreign language interface	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
194	Standard numerical libraries	2	2	2	2	2	2	2	1	1	2	1	2	1	1	1	0	0	2	0	2
195	Standard graphical software	0	2	2	2	2	2	1	1	1	1	0	1	1	0	0	0	0	1	2	1
196	Standard data–base interface	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
197																					
198																					
199	AVAILABILITY																				
200																					
201	Executional version available	2	2	0	2	2	2	1	1	2	2	2	2	1	1	2	1	1	2	2	1
202	Source code available	2	0	0	1	0	0	0	1	2	2	1	2	0	1	2	1	0	2	2	0
203	Public domain software	2	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
204																					
205																					
206																					
207	TOTAL # OF POINTS:	39	116	116	59	149	93	52	86	106	76	114	71	55	29	36	25	56	65	46	40

This notation is perfectly legitimate, but unfortunately it leads to confusion. For example, if one wants to add two polynomials: p(s) + q(s), this operation cannot be written as P + Q unless both polynomials p(s) and q(s) are of the same order. p(s)*q(s) is entirely different from any matrix multiplication; namely, it is a convolution between P and Q, e.g., expressed in CTRL-C as CONV(p,q). Thus, to be able to overload the conventional operators, we must be able to distinguish polynomials from other vectors through use of a different data type. (The term *operator overloading* has been borrowed from the ADA language. It denotes the ability to reuse the same operator for different purposes depending on the data types of its operands.)

Linear systems are expressed, e.g., in MATRIX_X, through an ordinary MATLAB matrix:

$$\underline{S} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

but certainly, two parallel connected subsystems cannot be expressed as S1 + S2, while two cascaded subsystems cannot be expressed as S2*S1, which would be analogous to frequency domain operations.

In IMPACT, each conceptual data element (matrix, polynomial, transfer function, linear system, trajectory) is expressed by a data structure of its own, thus allowing mathematical operators to be overloaded. This concept also allowed us to implement additional data structures such as polynomial matrices (which are essentially tensors) for the design of MIMO systems in the frequency domain (an area in which most CACSD tools are weak). Polynomials can, in IMPACT, be represented through their coefficients, their roots, or a set of supporting values.

Besides all these differences, the gap between the various MATLAB derivatives seems to close rather than to widen. Recently, a new downscaled version of MATRIX_X was announced for the IBM-PC. This software must obviously be in direct competition with PC-MATLAB. On the other hand, a new upscaled version of CTRL-C is currently under development that will execute on a VAX-Station II, and that will offer features very similar to the combined MATRIX_X/SYSTEM_BUILD software.

Columns 8–14 summarize other general-purpose CACSD programs. CATPAC (Buenz, 1986) is a strongly data-driven program. Both the numerical algorithms and the computed results are stored in hierarchically organized databases. Database management (cf. Chapter 7 of this book) is one area in which the MATLAB derivatives are chronically weak. The data organization in CATPAC eases the incorporation of additional algorithms into the program, and it allows results from CACSD studies to be maintained in a better-organized manner. CATPAC does not provide for a full-fledged language though; that is, new algorithms must be hand-compiled before they can be added to the database. Moreover, the execution of data driven programs is slower than the execution of code-driven programs. Thus, the runtime efficiency of CATPAC algorithms must lie somewhere between that of MATLAB derivatives when executing functions programmed in their own interpretive macro languages, which execute even slower, and that of MATLAB derivatives when executing algorithms programmed in their implementation languages (FORTRAN, C, ADA), which execute much faster.
Our experience with CTRL-C has shown that an algorithm programmed

directly into the CTRL-C system executes roughly 20 times faster than the same algorithm when implemented as a CTRL-C function (actually macro). The optimal solution would probably be to stay code-driven (as in MATLAB), but to provide a function translator that can compile new functions from the application language (MATLAB, CTRL-C, etc.) into the implementation language (FORTRAN, C, ADA), once such a function has been sufficiently debugged. This is the path we intend to take with IMPACT (which is one of the major reasons why language constructs in IMPACT's own procedural language look as similar as possible to equivalent constructs of its implementation language ADA).

Program CC (Thompson, 1986) is a CACSD program that has been implemented in BASIC and runs on IBM-PCs (and PC lookalikes). CC has been designed for multivariable control system analysis and synthesis. It supports the data type "transfer function matrix," which enables frequency domain design also of multivariable systems. Among the MATLAB derivatives, only IMPACT and M offer this feature. Special features of Program CC are graphical displays of transfer functions, partial fraction expansions, and inverse Laplace and Z-transforms, symbolic manipulation of transfer functions, and state-space and frequency domain analysis of multirate sampled-data systems. The symbolic manipulation of transfer functions is a rather unique feature. Only four programs of the 20 surveyed offer any symbolic manipulation capabilities at all. Such a capability is invaluable in parametric studies, and it is certainly one of the features that needs to be strengthened in the future.

KEDDC (Schmid, 1979, 1985) is one of the most comprehensive CACSD programs currently available. It is coded in portable ANSI-FORTRAN, with the exception of the graphics subsystem, which is coded in PASCAL. Interfaces (graphics drivers) exist for many different terminal types and brands. The user interface can be configured. KEDDC can be used in a question-and-answer mode, but is more flexible when used through its menu- and form-driven interface. Depending on the terminal type, KEDDC also supports window management with parallel sessions. Its user's manual alone has 1200 pages, its programmer's manual has 1400 pages! KEDDC thus takes the cureall approach rather than providing for a flexible control shell. Indeed, when the user wants to solve a problem for which no algorithm is currently provided in KEDDC, he may find KEDDC less user-friendly than some of the MATLAB derivatives. The combination of existing algorithms into new ones can less easily be accomplished. However when using existing algorithms, KEDDC offers better guidance, and in some cases offers better-tuned algorithms.

L-A-S (linear algebra and systems; Chow et al., 1983; West et al., 1985) is one of CACSD programs that offer a full-fledged programming language for implementation of new algorithms. L-A-S is thus the pure contrary to KEDDC. It follows the shell approach rather than trying to provide a cure-all. Unfortunately, we find the L-A-S language highly cryptic.

The LUND software system consists of a suite of different CACSD programs for various purposes. SIMNON (Elmqvist, 1975, 1977) is a direct-executing simulation language comparable in power to DESCTOP (Korn, 1985, 1987). It supports the simulation of sampled-data systems, the interconnection of submodels, and multirate integrations of subsystems. SIMNON was the first floating-point direct-executing simulation language. SIMNON executes on VAX/VMS and UNIVAC machines, and only a few months ago, a PC-version of SIMNON was added. SIMNON is coded in FORTRAN-77 (but unfortunately not in a very portable manner).

A preprocessor, DYMOLA (Elmqvist, 1978, 1980), a topology-oriented system description language, generates either SIMNON application programs or directly FORTRAN code that can be linked with the SIMNON system. DYMOLA models (subsystem descriptions) are connected by means of cuts, a new data abstraction mechanism (similar to PASCAL records) that allows grouping variables together in the same way as fibers are grouped into wires, and wires are grouped into cables. There are two versions of DYMOLA. The first version was coded in SIMULA; a newer version is coded in PASCAL. DYMOLA was the first modular modeling system developed.

A graphical prepreprocessor, HIBLIZ (Elmqvist, 1982; Elmqvist and Mattson, 1986), generates a DYMOLA program out of a graphical description of the model. This system was already mentioned in this appendix. To our knowledge, it was the first (experimental) graphical modeling system ever built. It executes on VAX/VMS using a modified CRT with a mouse. In the meantime H. Elmqvist, the designer of all three simulation/modeling systems, has left the Swedish Control Institute at Lund. With him, K. J. Åström lost one of the most talented and innovative simulation experts available.

IDPAC (Wieslander, 1980a) is a highly powerful program for parameter estimation in linear stochastic MIMO models and for model validation purposes. It provides for spectral analysis, correlation analysis, and data analysis in general. POLPAC (Åström, 1985) is a (somewhat more experimental) program for polynomial design. MODPAC (Wieslander, 1980b) is a program for transformations between different system representations; for basic operations on these data types, that is, polynomial operations and matrix operations; for evaluation of basic system properties such as controllability, observability, and stability; and for graphical output.

All programs contained in the Lund software suite use internally the same parser, INTRAC (Wieslander and Elmqvist, 1978), which also provides for a standardized (though somewhat primitive) "language" environment. Even though some of the MATLAB derivatives are more powerful than INTRAC with respect to the features offered through their MACRO languages, the LUND software suite has paved the road for this new software technology. The LUND software suite was the first modern CACSD tool on the CACSD software market.

TRIP (transformation and identification program; Van den Bosch, 1985) is a CACSD program for the design of continuous- and/or discrete-time SISO systems in either time or frequency domain. It has a tight link to the interactive nonlinear simulation program PSI for the design of nonlinear control systems. TRIP is available on VAX and IBM-PC. It is a low-cost CACSD program for the design of systems up to tenth order.

The Waterloo Control System Design Software Packages (WCDS and DSC; Aplevich, 1986) can be used for optimal-control, multivariable frequency domain design and algebraic matrix-fraction design for systems of up to 100th order. These software systems make use of the system representation $(\underline{F} - s\underline{E})\underline{x} + \underline{G}\underline{u} = \underline{0}$, where $\underline{E}$, $\underline{F}$, and $\underline{G}$ are matrices, u denotes the input vector, $\underline{x}$ denotes the state vector, and s is a linear operator. This is the direct linearization of the nonlinear vector equation $\underline{f}(\underline{x},s\underline{x},\underline{u}) = \underline{0}$. Any linear model in state-space form, transfer-function representation, or matrix-fraction form can be put into the above representation by inspection, and hence, in principle, any algorithm based on these other system representations can be implemented. However, important simplifications result from this choice of data representation. All binary algebraic operations required for manipulating subsystems are performed using exactly three operations:

combining two systems into one, adjoining linear constraint equations, and reducing a model to an externally equivalent model, with a desired form. The first two operations are numerically harmless; the third requires a generalization of state-space minimality theory and is implemented using numerically stable staircase algorithms.

Columns 15–21 of Table A.1 describe more special-purpose CACSD systems. Each of them [ICARE (Gorez, 1986a), MADPAC (Bartolini et al., 1983), PAAS (Gorez, 1986b), SANCAD (Gray, 1986), SSPACK (Technical Software Systems, 1985), SUBOPT (Fleming, 1979), and SUNS (Atherton and Wadey, 1981; Atherton et al., 1986)] has a strength in one particular area of CACSD, as can be extracted from the feature table. We shall not describe those software systems in any greater detail.

Although we added up the credit units for all these 20 CACSD systems, it is impossible to ensure that we really compared apples to apples in all cases. In particular, the special-purpose CACSD programs are obviously doomed to collect fewer points. Moreover, the question whether to assign 1 or 2 units to a particular feature is somewhat subjective, and the fact that a particular table column (CACSD program) contains many 1 entries may, in fact, speak for the modesty of the software designer, more than for a lack of quality of the particular product. Nevertheless, 5 of these 20 programs (MATRIX$_X$, CTRL-C, IMPACT, LUND, and KEDDC) collected more than 100 points each, and indeed, we believe these five programs to be the most versatile among the CACSD software systems surveyed. With its new link to SIMNON, PRO-MATLAB should be added to this list.

Our CACSD survey is obviously incomplete, as only about 50% of the CACSD software designers known to us by the time we mailed out our circular letter responded to our questionnaire. Another survey of CACSD software was published recently by D. K. Frederick (1985). In that survey, each software system is described in much less detail (as the survey does not contain any feature table), but the survey contains 37 entries in place of our 20, and these are only partially the same as those contained in our survey. Thus, Frederick's survey is highly recommended for an additional source of information (as is the entire volume in which that survey appeared). There has recently also been created an ELCS (extended list of control software) newsletter to provide an information exchange forum for CACSD software developers (Rimvall et al., 1985). The newest issue of this newsletter contains already more than 80 different software entries.

Our software survey is subjective rather than objective. This is due to the fact that the 1 and 2 entries could not be assigned in a completely objective manner. Moreover, the authors of this survey have a strong inclination in favor of MATLAB-like languages and did not hide this fact. We often expressed personal opinions rather than restricting our survey to a simple listing of dry facts and leaving the interpretation of these facts to the reader, but we tried honestly to serve the CACSD community in the best possible way. The authors acknowledge with gratitude the kind cooperation of those responding to our questionnaire.

A.5 STANDARDIZATION VERSUS DIVERSIFICATION

In the previous section, we introduced a number of different CACSD tools. We have seen that they vary with respect to their application areas, as well as with respect to their user interfaces. Is such a diversification really justified and desirable, or might a canalization of the various and diversified efforts into one CACSD software standard be more appropriate? Is there any hope for a CACSD standard comparable, for example, to the continuous system simulation language (CSSL) standard in simulation software (Augustin et al., 1967)? What might such a standard look like?

We indeed believe that with respect to the manner in which control problems are formulated, a standard is both feasible and desirable. 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 a few publications have used this notation for simplicity. Hopefully, a MATLAB-like notation will also be introduced into CSSL's as an additional tool for the description of state-space models.

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 where 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

$$\underline{P} = \left[\begin{array}{cc} (3s^2 + 10s + 3 & (2s - 3) \\ s^3 & (-s^2 - 7s - 10) \end{array} \right]$$

can, in IMPACT, be coded as

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

or alternatively as

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

to denote the factorized form

$$\underline{\underline{PF}} = \left[\begin{array}{cc} 3(s + 3)(s + 1/3) & 2(s - 1.5) \\ s*s*s & -(s + 2)(s + 5) \end{array} \right]$$

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, e.g., 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) will result in an error message. This notation has meanwhile been adopted by the developers of M as well (Gavel et al., 1985).

Of course, it is natural to define once and forever

```
s = [ ^1 ]
```

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

```
p1 = 3*s**2 + 10*s + 3
```

or alternatively as

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

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

```
sf = [ |0]
```

and thereafter

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

which, however, is not recommended as frequent defactorizations and refactorizations will take place in this case. Notice 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 CTRL-C, 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 = 1, I = 3, I = 7, and I = 28, respectively. (Many users still deplore the missing GOTO statement in CTRL-C. Although it has been theoretically proven that a GOTO statement is not needed, its lack makes programming sometimes quite difficult.) 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 Swiss mouse is a very convenient, flexible, and fast-input device, and it is expected to make the previously fashionable cross-hair cursors and light pens soon obsolete, as cross-hair cursors are both uncomfortable to use and slow, and as light pens demand very expensive screen sensors. To our displeasure though, there exist mice with one, two, and 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 (like driving a car that has the gas pedal on the left and the clutch on the right).

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, as too early a standard can freeze the lines and hamper the introduction of innovative new concepts.

Another interface, which is rarely even noticed by the casual CACSD software user, is the interface to a database where results of computations, as well as programming modules, notebook files, etc., 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 don't even offer a database interface at all, but fully rely on the file-handling mechanism (directory structure) of the embedding operating environment. This mechanism is computationally efficient (as 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 have previously been stored for later reuse. As an example, a particular A-matrix of a linear system will probably not be related to the problem under investigation at all, but will be 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).

An IFAC working group discussing "Guidelines for CACSD-Software" was generated recently which consists of three subgroups for the discussion of

CACSD program interfaces (including graphics)
CACSD program data exchange
CACSD program algorithm exchange

A similar IEEE working group exists as well. Hopefully, these two bodies will be able to promote a forthcoming CACSD standard.

A.6 SIMULATION AND CACSD

Let us discuss next how simulation has been implemented in some of the current CACSD programs.

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

$$[\underline{y},\underline{x}] = \mathrm{SIMU}(\underline{a},\underline{b},\underline{c},\underline{d},\underline{u},\underline{t})$$

where $\underline{a}$, $\underline{b}$, $\underline{c}$, and $\underline{d}$ are the system matrices describing a linear continuous-time MIMO system, $\underline{t}$ is a time base (that is, $\underline{t}$ is a vector of time instants), $\underline{u}$ is the input vector sampled over the time base (that is, $\underline{u}$ is actually a matrix, each row denotes one input variable, and each column denotes one time instant), $\underline{y}$ is the output vector (that is, $\underline{y}$ 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',x0)
```

and also the integration method can 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 out of subsystem descriptions by use of a series of interconnection functions (SERIES, PARALLEL, INTERC, and MINREAL).

In IMPACT, we chose a slightly different approach. Since 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):

```
t = LINDOM(0.,1.,0.1) & LINDOM(2.,20.,1.) & 50. & 100
```

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:

```
u = [SIM(t);COS(t)]
```

which creates a trajectory column vector u evaluated over the previously defined domain t. Linear systems are generated by use of the LINCONT and LINDISC functions:

```
s1 - LINCONT(a1,b1,c1,D=>d1,X0=>[0.5;2.;-3.7])
```

The three matrices a1, b1, and c1 are compulsory positional parameters, whereas the input-output matrix (D) and the initial condition vector (X0) are optional (defaulted) named parameters.

Series connection between two subsystems is expressed as s2*s1, that is, multiplication in reverse order (exactly what it would be if the two subsystems were expressed through two transfer function matrices: g2*g1); parallel connection is expressed by use of the "+" operator, and feedback is expressed by the "\\" operator:

```
gtot = g \\ (-h)
```

(g fed back with -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:

```
y = s1*u
```

which simulates the system s1 (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:

```
tout = (s2*s1)*tin
```

series-connects the two subsystems s1 and s2 and then performs one simulation over the combined system. On the other hand:

```
tout = s2*(s1*tin)
```

simulates the subsystem s1 using tin as input, samples the resulting output trajectory over the same domain, and then simulates the subsystem s2 using the previous result as an input by reinterpolating it between its supporting values. Of course, numerically the results of these two operations will be 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 an embedding into a larger whole in which simulation is just 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 just 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 of type 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 tout = s2*(s1*tin) 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 taux = s1*tin) would have to be manually edited into a "tabular function" and used by the second simulation run (implementing tout = s2*taux).

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 $\underline{K}(t = t_f) = \underline{0}$, and because the Riccati equation is numerically stable in backward direction only. The solution $\underline{K}(t)$ is stored away during this simulation and then reused (in reversed order) during the subsequent forward integration of the state equations with given $\underline{x}(t = t_0)$. Some of the available CSSLs allow solving this problem (mostly in a very indirect manner); other 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. In the newest release of CTRL-C, an interface to the well-known simulation language ACSL was introduced. This interface allows making use of the modeling and simulation power of a full-fledged simulation language, while one is still able to control the experiment from within the more flexible environment of the CACSD program. Several of the discussed CACSD programs 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

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

Since ACSL does not provide for a powerful matrix environment, we have to separate this compact matrix differential equation into its component equations. [ACSL does provide for a vector-integration function, and matrix operations such as multiplication and addition could be (user-)coded by use of ACSL's MACRO language. However, this is a slow, and inconvenient replacement for the matrix manipulation power offered in languages such as CTRL-C.] Furthermore, since 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

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

forward in time from $t^* = 0$ to $t^* = t_f - t_0$. Through the new interface (A2CLIST), we export the resulting $K_{ij}(t^*)$ back into CTRL-C, where they take the form of ordinary CTRL-C vectors. Also in CTRL-C, we have to manipulate the components of $\underline{K}(t)$ individually, as $\underline{K}(t)$ 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). Back substitution can be achieved conveniently in CTRL-C by simply reversing the order of the components of each of the vectors as follows:

```
[n,m]  =  SIZE(kij)
   nm  =  n*m
  kij  =  kij(nm:-1:1)
```

Now, we can set up the second simulation:

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

What we would like to do is to ship the reversed $K_{ij}(t)$ 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 $K_{ij}(t)$ 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 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 due to the fact that ACSL is not (yet!) sufficiently powerful for our task, and that 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 out 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 of the fact that (as the above example shows) the currently available simulation languages are really not very well suited for our task, and partly due to our decision to employ ADA as implementation language. As currently no CSSL has been programmed in ADA, we would have had to rely on the "pragma concept" (which is ADA's 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 as 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 was demonstrated. However, nonlinear systems can be coded as special macros (called SYSTEM MACROs). The two linear system types (LINCONT and LINDISC) are, in fact, just 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;
```

Notice 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 with 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 case of 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 the above example, the entire integration information, in IMPACT, is stored in the domain variables, which makes sense as these variables anyway contain part of the runtime information (namely, the communication points and the final time). Moreover, 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 not yet been released. Roughly the first 75,000 lines of ADA code have meanwhile been coded and debugged, and the IMPACT kernal will be released soon. This kernel will implement all the IMPACT language structures (including all the macro types, the complete query feature, and multiple sessions), but it will not contain all the foreseen control library functions, nor will it contain multiple windows. The complete software is expected to become available soon.

A.7 OUTLOOK

How is the field of CACSD going to 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 partly determine 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 own special-purpose languages. The difference is simply that less and less of the computational task is frozen in code, while more and more of it is parameterized, that is, data-driven. The data in itself reached such a degree of complexity that its 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 not 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 new direct executing (that is, fully data-driven) simulation languages such as SIMNON (Elmqvist, 1975, 1977) and DESCTOP and DESIRE (Korn, 1985, 1986), 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. These types of simulation tools are exactly what is needed within the CACSD program.

However, we are currently at the edge of taking yet another step. We now talk about the development of multiwindow user interfaces, of object-oriented programming style, of language-sensitive editors, of CAD databases, etc. 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 the fact 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 do believe that future programming systems will blur the previously clear-cut distinction between programming languages and the operating systems they are embedded in. 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 it is to run under. 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, as the ADA environment itself is not yet completely defined, and as we would like to borrow as much as possible from ADA concepts. In M (Gavel et al., 1985), 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.

The future will tell how efficient (or rather inefficient) EAGLES is when implemented as a language running under an existing operating system (as it is currently planned) rather than being implemented as the operating system itself. Obviously, EAGLES has to rely strongly on the record manager and system functions of the operating system (VMS). How will the developers of EAGLES deal with new releases of the underlying operating system offering enhanced and, at this level, not fully upward-compatible new features? Are EAGLES users going to have the same problem as EUNICE users (EUNICE = UNIX under VMS), who are always two or three versions behind the current version of VMS, because it takes the EUNICE developers usually 1 year to keep up with the newest (meanwhile already again outdated) developments in VMS? We don't know the answers to these questions. We just focus on some of the dangers behind this current development.

One way to overcome the previously mentioned problem 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 (unfortunately not very uniform) UNIX implementations for various computers. The idea is splendid. Unfortunately, the original UNIX was much too small to be taken seriously (e.g., with respect to the problem of enforcing data security and data integrity). Current UNIX dialects tackle these problems in various ways, but the experiment is doomed to failure unless the computer manufacturers can agree on an invariant UNIX kernel that goes far beyond the original UNIX definition. It must include not only the procedural command language, but also system calls (lexical functions), the interface to the record manager, and naming conventions for files, symbols, and logicals.

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 of data-driven programming is a language in which there is essentially no longer any difference between code and data at all. 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, while all further entries are its parameters. For these reasons, LISP programs exhibit a serious runtime inefficiency. A numerical algorithm implemented in LISP will probably execute two to three orders of magnitude slower than the same algorithm implemented in a conventionally compiled language. Moreover, LISP is often rather unwieldy with respect to how a particular numerical algorithm has to 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, numerical data are entries in lists just like any other data. Thus, nonnumerical data processing is as efficient as numerical 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 as compared to 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 efficiency of executing LISP primitives. Such machines are already available and help to overcome at least part of the inefficiency of LISP.

With respect to the user interface, many of LISP's difficulties can be avoided by changing the world view once more. While LISP is basically process-oriented, 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 than even LISP. (However, PROLOG can rather easily be implemented in LISP, and thus, there exist PROLOG environments also on LISP machines, and they function amazingly well.) PROLOG primitives are more compound than LISP primitives. The natural consequence 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, while the converse is true.

These new languages are expected to shortly lead to yet another generation of CACSD tools. The strength of these new tools will lie in nonnumerical design, that is, in parameterized control system design studies, where several parameters are kept as unknowns in the design process. This will hopefully help to give the designer more insight into what is happening in his system. Nonnumerical controller design algorithms are still in their infancy, and it is not known yet how far these new concepts will lead us. How does one avoid the problem of formula explosion; that is, how can one obtain parametric answers without being confronted with pages and pages of never-ending formulas? Recent developments in programs such as MACSYMA (Symbolics, 1983) and REDUCE (Rand Corp., 1985) may help to answer some of these questions. A clue may be to introduce intermediate variables in the right places, variables that are not further expanded but kept as additional (dependent) parameters. These techniques were recently surveyed by Birdwell et al. (1985).

Another area that will be boosted by concepts such 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 nonnumerical 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 above 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, graphical, 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
```

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 has to check 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. And finally, if a is a rectangular matrix, x is evaluated as the solution of an over- or underdetermined 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, there is more expert system technology readily available than what is commonly exploited. To give an 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, that does not mean that they have to be processed within the expert system in a symbolic form as well. Compiler writers have known this fact 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 numerical quantities in place of the symbolic ones. Only upon output (e.g., for generating the cross-reference table), the original symbols are retrieved through the hash table. This mechanism could easily be used in expert systems to increase their efficiency, but this is rarely done today. SAPS (Uyttenhove, 1979; Klir, 1985; Yandell, 1987), for instance, can be used for qualitative simulations of discrete input/output models (that is, models described through sets of input and output trajectories rather than by means of 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≫, and ≪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." However, in SAPS, the recoding has to be done by hand, and the output will be expressed in terms of the recoded variables instead of the original ones.

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 which may suffice for the expert, but is inadequate for the novice. Thus, after receiving such a message, the user can type a "?" which is honored by a more detailed analysis of the problem. IMPACT's QUERY facility is another step into this direction. Another implementation has been described by Munro et al. (1986).

Also, K. J. Åström and L. Ljung are working on such a facility for IDPAC (private communication). The idea is the following: Rather than letting the students 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 (CAI) 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 K. J. Åström (private communication). He wants 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 on not only for syntactic correctness, but also for semantic correctness. Some of the semantic tests are, of course, data-dependent, and these can only be performed 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, e.g., Gaussian elimination. 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, etc. 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, a 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 so good for global assessment of complex systems. A complex system such as the forthcoming space station or a nuclear power plant needs to be monitored, and expert system technology may be used to decide when something odd has happened or is about to happen. Then a global control strategy must take over and decide what to do next. Currently, human operators do a much better job in this respect that 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. Cellier (1986b) investigates the possibilities of qualitative simulation and rule-based control system design.

To sum up, CACSD is still a very active research field, and more results are to be expected shortly. We sincerely hope 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*. Zürich, Switzerland, August 29–31, 1979 (M. A. Cuénod, ed.), Pergamon Press, Oxford, 441–446.

ANSI (1985). *American National Standard for Information Systems, Computer Graphics, Graphical Kernel System (GKS)*. Functional Description (ANSI X3.124–1985) and FORTRAN Binding (ANSI X3.124.1–1985).

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, 266 pp.

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.

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 Publishing, 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. et al. (1986). SUNS: The Sussex University Control Systems Software, in: *Proceedings of the 3rd IFAC Symposium on Computer-Aided Design in Control and Engineering Systems (CADCE'85)*, Copenhagen, July 31–August 2, 1985, Pergamon Press, pp. 173–178.

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.

Bartolini, G., et al. (1983). A package for multivariable adaptive control, in: *Proceedings of the 3rd IFAC/IFIP Symposium on Software for Computer Control (SOCOCO'82)*, Madrid, Spain, Pergamon Press, Oxford, pp. 229–235.

Birdwell, J. D. et al. (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–August 2, 1985, Pergamon Press, Oxford, pp. 1–8.

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

Cellier, F. E. and Rimvall, M. (1983). Computer-aided control systems design, Invited Survey Paper, in: *Proceedings of the Winter Simulation Conference (ESC'83)*, Aachen, FRG (W. Ameling, ed.), Springer-Verlag, Lecture Notes in Informatics, New York, pp. 1–21.

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–April 1, 1977, Pergamon press, pp. 151–159.

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.

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

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, 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, 226 pp.

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 (SOCOCOCO'82)*, Madrid, Spain, Pergamon Press, Oxford.

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

Evans, D. C. (1985). The art of visual simulation, Keynote Address, 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*, Zürich, Switzerland, August 29–31, 1979 (M. A. Cuénod, ed.), Pergamon Press, Oxford, 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 systêmes. 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.

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.

Kailath, T. (1980). *Linear Systems*, Prentice-Hall, Englewood Cliffs, NJ. 682 pp.

Klir, G. J. (1985). *Architecture of Systems Problem Solving*, Plenum Press, New York, 539 pp.

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 and Simulation 8(4).

Korn, G. A. and Wait, J. V. (1978). *Digital Continuous System Simulation*, Prentice-Hall, Englewood Cliffs, NJ. 212 pp.

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.

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

Little, J. N. et al. (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, Springer-Verlag, New York.

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.

Mitchell, E. E. L. and Gauthier, J. S. (1986). *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. 40 pp.

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

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.

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.

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

Pegden, C. D., et al. (1985). *CINEMA User's Manual,* Systems Modeling Corp., State College, PA.

Rand Corp. (1985). *REDUCE User's Manual,* The Rand Corp., 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, 208 pp.

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–August 2, 1985, Pergamon Press, Oxford, 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., et al. (1985). *ELCS—Extended List of Control Software, Newsletter* (M. Rimvall, D. K. Frederick, C. Herget, and R. Kook, eds.), 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, adv.), Dept of Automatic Control, ETH-Zentrum, Zürich, Switzerland.

Schmid, C. (1979). *KEDDC, User's Manual and Programmer's Manual,* Dr. -Ing. Chr. Schmid, Lehrstuhl für Elektrische Steuerung und Regelung, Ruhr University Bochum, Federal Republic of 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., et al. (1986). *TESS with SLAM-II, User's Manual, Version 2.2,* Prisker & Associates, Inc., West Lafayette, IN.

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.

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.

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, 40 pp.

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, 108pp.

Wieslander, J. (1980b). *MODPAC Commands—User's Guide,* Report: CODEN: LUTFD2(TFRT-3158), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden, 81 pp.

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, 60 pp.

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.

Yandell, D. W. (1985). *SAPS-II: Raw Data Analysis in CTRL-C, User's Manual and Progress Report,* Senior Project (F. E. Cellier, adv.), Dept. of Electrical and Computer Engineering, University of Arizona, Tucson, AZ.