Interaction in Computer Aided Analysis and Design of Control Systems

Wieslander, Johan

1979

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Wieslander, J. (1979). *Interaction in Computer Aided Analysis and Design of Control Systems*. [Doctoral Thesis (monograph), Department of Automatic Control]. Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:
1

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

CODEN: LUTFD2/(TFRT-1019)/1-222/(1979)

Interaction in Computer Aided Analysis and Design of Control Systems

JOHAN WIESLANDER

LTH

Department of Automatic Control · Lund Institute of Technology

Johan Wieslander

Interaction in Computer Aided
Analysis and Design of Control Systems

Interaction in Computer-Aided Analysis and Design of Control Systems

av

Johan Wieslander

Tekn lic, Ld

Akademisk avhandling som för avläggande av teknisk doktorsexamen vid Tekniska Fakulteten vid Universitetet i Lund kommer att offentligen försvaras i sal M:E, Maskinhuset, Lunds Tekniska Högskola, måndagen den 21 maj 1979 kl 10.15.

Dokumentutgivare
Lund Institute of Technology
Dept of Automatic Control

Handläggare
Karl Johan Åström

Författare
Johan Wieslander

Dokumentnamn
REPORT

Utgivningsdatum
May 1979

Dokumentbeteckning
LUTFD2/(TFRT-1019)/1-222/(1979)

Ärendebeteckning

Dokumenttitel och undertitel

Interaction in Computer Aided Analysis and Design of Control Systems

Referat (sammandrag)

The thesis discusses the use of Interactive programs to solve common problems of analysis and design of Control Systems. The two main interaction methods, command dialogue and question & answer dialogue are compared in the light of program structure and interaction needs. A program module that handles the man-machine communication is described. It realizes an interaction language with a macro facility, which is demonstrated to provide the means needed for interaction adaption. The role of data structures is explored, and here the possibilities of a Simula67 extension are made apparent. Finally, some demands on the ideal Interactive programming language are sketched.

Referat skrivet av
Author

Förslag till ytterligare nyckelord

Klassifikationssystem och -klass(er)

Indextermer (ange källa)
Interactive computing, Man-machine interface, Computer-aided design, Automatic control, Computer language.

Omfång
222 pages

Övriga bibliografiska uppgifter

Språk
English

Sekretessuppgifter

ISSN

ISBN

Dokumentet kan erhållas från
Department of Automatic Control
Lund Institute of Technology
Bos 725, S-220 07 LUND 7, SWEDEN

Mottagarens uppgifter

Pris

Interaction in Computer Aided Analysis and Design of Control Systems

Johan Wieslander

Lund 1979

To Boel

who patiently asked when

CONTENTS

1. INTRODUCTION

Automatic Control emerged as a separate discipline in the late forties, when the great advances in servo mechanism design made during the 2nd world war was made public. The methods employed were largely based on the Laplace transform and frequency response methods. They were largely restricted to single input - single output systems but were otherwise insensitive to the complexity of the system. The great advantage was that the methods were ideally suited for graphic representations. This allowed much of the design to be done using diagrams, paper and pencil. Special purpose templates were available to aid in the drawing of curves, further decreasing the amount of computations needed. The techniques developed in the forties and fifties are still adequate for many problems encountered in control applications.

The introduction of the general purpose digital computer had a dramatic impact on control theory. One consequence was that a numerical solution to problems became both feasible and acceptable. This in turn opened new directions for research. The result was an expanding theory for control systems based on a description in the time domain. Noteworthy examples are the linear-quadratic optimal control theory and Kalman filtering theory. Among other important characteristics are that they allow a solution to problems with several inputs and several outputs. One decade later, in the late sixties, generalizations to the frequency response methods to the case with multiple inputs and outputs became known. Common to all these methods is that their application to problems other than mere textbook examples leads to substantial numeric computations.

Thus the computer enters into the life of the control engineer as the most important design tool. Important questions are then raised: What are the needs of the engineer? How can the powers of the computer be made available to him satisfying these needs?

Computer use

The normal use of a computer for design purposes in the sixties was in the form of batch computation, i.e. a problem and the intended way of its solution were formulated entirely and in advance. This information was then fed into the computer through some queuing system. After some time, ranging from minutes to several hours (or even days), the result was available.

The use of a computer in batch mode was forced on the design engineer by economic factors such as efficiency in the use of expensive equipment. Some acute drawbacks of this way of using a computer were the need to plan every detail in advance, the idle hours in waiting for the result, and not least the sensitivity of the procedure for simple errors in the input data. A single misplaced digit or other typing error resulted in the loss of time and money.

The principal and fundamental drawback with batch computing was that it did not exploit the important and complementary qualities of computer and man. The computer has great ability for computing and data handling. Man is good at things like using experience and prior knowledge in decision making, detecting patterns in results and generally applying "common sense". The combination of these good qualities of man and machine would have a great potential in any application field. This is the goal in the design of interactive programs.

Interaction

When computer power began to be available either through time-sharing terminals or through mini- or midi-computers run in open shop, there was an opportunity to realize the desires for a closer interaction between computer and man. A natural first step was to include guiding questions in the input phase of the programs to avoid the annoying errors caused by mistakes in order or format of data. The second step would then be to show results as they become available together with a list of possible next steps. The user could then direct his future actions based on the results of previous steps.

This approach which is called <u>question & answer interaction</u> is very common and has brought great advances in computer use into many application fields. Although it meant a great step forward, some properties of this interaction method may be regarded as drawbacks, at least in some cases. This has lead to an alternate approach, viz. <u>command interaction</u>. This has different properties, with advantages and disadvantages.

The discussion of typical interaction needs, how they may be satisfied and what problems that have to be solved, are the main topics of this thesis. Along with this material of a general nature, there will be specific examples taken from programs and program modules in actual existence.

The background

The background for this thesis and the source of the experience and results reported, is a project at the Department of Automatic Control, Lund Institute of Technology, Sweden. The goal of the project is to make common methods of design and analysis available to the control engineer in the form of interactive programs. The result so far is:

a) Idpac, for identification and data analysis.

b) Synpac, for synthesis of multivariable systems.

c) Modpac, for analysis and transformation of models.

d) Polpac, for design of single output systems on transfer function form.

e) Simnon, the result of a parallel project, for simulation of nonlinear systems.

All of these programs are identical in structure and use the same library of support software, including the interactive communication module Intrac. The first three are used in the examples in Chapter 7 & 8. Their command lists are included in the appendix.

Some of the programs have been available for a long time and are used routinely by people outside the department to solve real-life problems. They have also been exported on a commercial basis.

An outline of the work

Chapter 2 is a survey of some computational needs encountered in Automatic Control. The presentation is based on a hypothetical design process starting with measurements to obtain parameter values, the actual estimation of such parameters, analysis and synthesis stages, simulation and implementation of control strategies. The aim of this chapter is to show that the needs for interaction are more pronounced in some areas than in others, and that interaction may play an essential role.

Chapter 3 starts by extracting the typical needs on interaction found in Chapter 2 into a more application independent form. Then the two main contenders, the question & answer dialogue approach and the command dialogue approach are scrutinized and compared, also in the light of programming requirements. Finally, the interactive user is studied and found heterogeneous. This gives rise to the question of which type of user to have as reference in the design of interactive programs. The discussion of this matter concludes Chapter 3.

Chapter 4 is a presentation of the communication module Intrac used ina the interactive programs mentioned above. Intrac imposes a certain structure both on programs based on this module and on command lines directed to such programs. These structures as well as the internal structure of the module itself are discussed. Intrac efectively defines a language for interactive man-computer communication. The properties of this language is examined and some of its statements are defined. Intrac offers a possibility to dynamically define macro commands. This facility is shown in the final section to allow the adaptation of the interaction form to different levels of user needs.

Chapter 5 starts with a discussion on the different data types encountered in programs concerning control systems. It is demonstrated that in system descriptions there is a strong desire to be able to reference data of different internal structure. This requirement is possible to satisfy in programming languages like Simula, and the possibility to use this as a basis for interaction is briefly explored. The rest of the chapter describes how these problems were solved in down-to-earth FORTRAN.

Chapter 6 is the last one. It gives a historical account of the development that resulted in this thesis. As side effects, experiences with interactive hardware as well as software problems are discussed. As a conclusion, some thoughts on a desirable interactive programming language are given.

Finally, there are two examples. They are intended to give some flavour of the use of the programs referred to in this thesis. Example 1 describes a laboratory process and a measurement experiment performed on it. The result is used to identify a model for the process. The example shows the use of a straightforward command dialogue.

Example 2 describes the design of an output feedback for a system taken from literature. The convenience of special purpose user defined macros is obvious in this example.

A concise description of available commands in the programs used for the examples is included as an appendix.

Acknowledgements

The thoughts and experiences reported here are results of a project extending over several years, supported by the Swedish Board for Technical Development under contracts 73-3553, 75-3776 and 77-3548.

Many people, directly or indirectly associated with this project, have contributed to its success. Ivar Gustavsson with his great expertise in identification problems has played a significant part in the design of Idpac. The same should be said about Gunnar Bengtsson, Lars Pernebo, and Ann-Britt Nilsson regarding Modpac and Synpac. The many ideas of Hilding Elmqvist have influenced the command module Intrac and much programming methodology. He also co-authored

[Wies78] from which much of the material in Chapter 4 was taken. Tomas Schönthal and Tommy Essebo did much of the programming work and were responsible for the many implementational details.

A most important factor has been the never ending interest, support, and stream of suggestions and good advice from my thesis supervisor prof. Karl Johan Åström. Noteworthy is his ability to create a nice athmosphere at the department, which has made the work possible and worthwhile.

Eva Dagnegård typed the manuscript with great interest and Britt-Marie Carlsson was very helpful in the preparation of the figures. Leif Andersson kindly read the entire manuscript, finding many errors. The access to a high quality printer at the Lund Computing Centre is also gratefully acknowledged.

2. COMPUTING PROBLEMS AND AVAILABLE METHODS IN CONTROL SYSTEM DESIGN

In this chapter we will view the task of control system design through the various possible steps from measurement (or experimental) phase to the implementation phase. Our main interest will be in locating those points where a computer may be essential, what methods are available, and not least, the requirements on the man-machine interaction. In this way, we will gather restrictions, requirements, and criteria for the design of interactive programs for computer aided design.

Some if not all of the following steps are likely to be taken in the path that leads up to a working control system. A model of the process to be controlled is likely to be used in the design phase. This model could be obtained through system identification or from model building based on basic physical principles. Also in the latter case, experiments may have to be done to obtain parameter values. Thus, a measurement step is likely to be used in the early stages of a design.

Having obtained measurement data the first task is to do a preliminary analysis to locate obvious errors, to perform scaling and calibration etc. Dynamical properties can then be estimated by non-parametric methods (correlation analysis, spectral analysis etc.) or by identifying a parametric model. An alternative is to use known or measured physical properties to build a model. Doing this, analysis and simulation of the entire model or part of it can be of much help.

Prior to the design phase, the designer will try to obtain a feeling for the relevant properties given by the model. This can be achieved through analysis, simulation, or transformation of the model to various alternate forms. The actual control design can be carried out using a variety of methods, depending on the complexity of the problem and the degree of detail given by the available model. Finally, the implementation of the control strategy may in some cases be included in this chain.

Thus measurement, identification, analysis and synthesis will be the areas of computer use studied in this chapter.

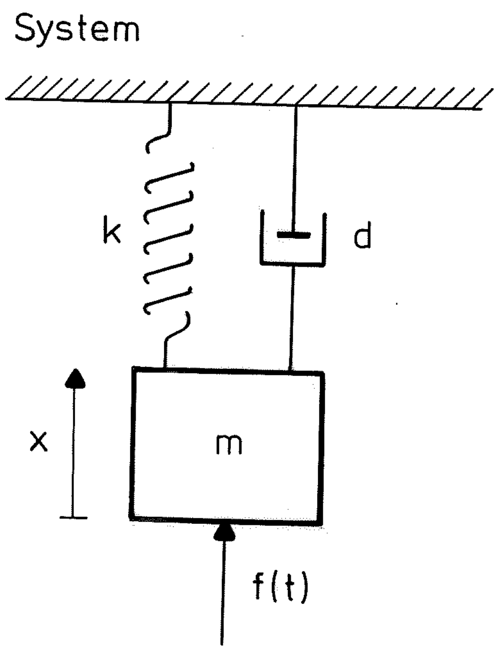
2.1 Measurements

Through measurements in a controlled experiment situation, it is often possible to gain the partial knowledge of the process that is essential in many control design methods. The type and degree of this knowledge determines which path to follow in the design. In some cases we know things in advance, through experience, general knowledge of physics and chemistry etc, or through component specifications. If we intend to use this prior insight in the form of model building, material constants, physical dimensions, heat transfer coefficients etc, are likely to be used. The need to measure such parameters thus arises.

Static and Dynamic Measurements

An experiment to determine parameters often consists of keeping some variables at a number of different fixed values and measuring one or several dependent variables. The coefficient or coefficients of interest are then obtained through some curve fitting method. Practical problems that may arise includes the need to keep the independent variables constant, and to wait until things have settled before the actual measurement takes place. We might call this a static measurement situation. In some cases though, this implies some sort of regulation, making the problem semi-dynamic in nature. Figure 2.1 gives an example of a system where some parameters are directly measurable with a static experiment.

A dynamic measurement situation is when the system is excited with time-varying test signals, and when both inputs and outputs are recorded as functions of time. The choice of input signals will depend on the expected properties of the system; typical are PRBS-signals, sine-waves and earlier measured and recorded plant signals. This case is also indicated in Figure 2.1. A dynamic measurement followed by some identification step will give the system parameters although in another form.



Equations

$$m\ddot{x} + d\dot{x} + kx = f(t)$$

or

$$\ddot{x} + 2\xi\omega\dot{x} + \omega^2 x = K \cdot f(t)$$

$$\omega = \sqrt{\frac{k}{m}} \qquad \xi = \frac{d}{2\sqrt{mk}} \qquad K = \frac{1}{m}$$

Figure 2.1. A static measurement could for instance be: Apply $f(t) = f_\emptyset$, observe $x(\infty) = x_\emptyset$, then $k = f_\emptyset / x_\emptyset$.
A dynamic measurement situation: Apply time-varying $f(t)$, observe $x(t)$, then use identification techniques to obtain ω, ξ, and K.

A practical problem that often arises is that the test signals may have to be (partly) the output of a regulator in order to keep the system under test stable or within safe operating conditions. On an industrial plant in normal operation, a requirement that production is not to be disturbed is natural. This gives restrictions on the test signals. In conclusion, dynamic measurements are made when at least part of the plant is in operation. For the ultimate control design to be relevant, the conditions must be as close to normal as possible.

Measurement Problems

A number of operations have to be performed during the different stages of the measuring experiment. First the channel assignments must be defined, i.e. the correspondence between variables and the hardware addresses in the measuring equipment. Likewise, input range and scaling factors have to be chosen.

After this, the hook-up has to be tested, to gain confidence that it is indeed the intended quantities that are obtained. Also, transducers have to be tested and maybe calibrated. Further, it is often useful to be able to document the conditions and parameters of an individual experiment, e.g. which of the variables that are actually recorded. The latter need stems from the fact that more variables may be connected to the measuring equipment than are actually interesting in a certain experiment.

During the experiment several facilities may be of interest. Useful features are the ability to inspect data visually, on a chart recorder or a computer's graphical display unit, simultaneously with their aquisition. In this way, obvious errors can be detected at once and the experiment can be repeated. If data is to be used to estimate process model parameters, at least some methods allow this to be done in real time. If this is done, the experiment can be terminated conditionally, when the desired accuracy has been reached. This may also be a method of detecting outliers, i.e. single measurement errors.

Finally, during the experiment, data should be recorded in a format suitable to the following analysis tools.

Use of a Computer

Special purpose equipment is available on the market that performs some of the operations mentioned above. On the measurement side, data loggers are common and can handle basic checkout and data recording operations. Correlation- or frequency analysers are useful to determine process dynamics.

Most modern equipment of the kind mentioned above is built around a mini- or micro-computer, maybe with some operations implemented in firmware. This field is currently growing very rapidly. Of course, a general purpose mini- or midi-computer could be used as well, provided suitable software and the necessary computer process interface hardware were available. A general purpose computer will not be as fast in such operations that the special purpose equipment was designed for, but is more flexible.

In many cases, the final control design will be implemented using a computer, maybe in a hierarchical configuration with many small slaves. Using this computer for the measurements ensures that the results reflect the situation where the control is going to be used. (E.g., time constants in transducers will be accounted for in the models.) A computer will also be tremendously useful in the intermediate steps. The use of a general purpose computer for the initial measurement phase will make the compatibility of programs and data more easy to attain. Indeed the same computer might be used for the computer aided measurement, analysis and design steps, provided it is adequately equipped.

Interaction

The aim so far has been to demonstrate the multitude of measurement tasks and problems, our interest being the interactive use of computers. Given that we choose to use a general purpose computer to satisfy our needs, what demands do we have on the interaction with the programs?

Clearly, although the path through the various stages of testing the hook-up, calibration, selecting variables, doing the measurement and documenting the experiment is generally applicable, the number of possibilities left open at each stage is large. Furthermore, measurement experiments may be performed relatively infrequently. These two observations point towards interaction methods with a high level of user guidance.

In repeated experiments, a number of operations are usually performed in a certain sequence with no or small alterations between times. This situation contrasts with the initial setup of a measurement series, when testing, channel selection etc. are being done, and when improvisations are common. All in all, the measurement situation exhibits a wide variety of interaction patterns. There are conflicts between desires for flexibility, guidance or for fixed sequencies of operations for standard problems. Figure E1.5 in Example 1 shows how this problem was solved in a certain implementation.

2.2 Preliminary Analysis of Measured Data

After data have been measured and recorded, the analysis phase follows. The first preliminary step serves to validate data and to perform proper scaling and adjustment operations. Data validation may e.g. be done by visual inspection of a plot of the data. In this way isolated measurement errors can be detected. Correction can either be by eye or by some interpolation formula. Automatic detection of measurement errors (outliers) could be done by various techniques such as (adaptive) filtering with tests or interpolation with tests.

The next thing to do might be to convert the measured values to engineering units. At the same time a calibration value is subtracted. In some instances a more complicated operation should be performed. An example is the removal of the effect of a known non-linearity in the measurement transducer. Another one is correction for known interdependencies between measurements, e.g. temperature compensation of a flow measurement. See references [Jens76] and [Hall78].

Following this phase, the data may have to be prepared e.g. for a following identification step. Identification methods are, at least in some cases, sensitive to DC-levels, or trends, in the measurements. Thus, trend estimation and removal is an operation that will be used here. Access to digital filters may also be useful.

Finally, statistical data such as mean, variance, largest and smallest value etc, may be of interest. Also more special qualities such as amplitude distribution or level durations may be desired.

Interaction

In general, operations to be performed in this area are largely determined from time to time by the operator, drawing on his experience and ability to detect patterns, as e.g. in the detection of "curious points" (errors) in a measured signal.

Some of the possible actions are likely to be ready made, such as displaying of data, modifying and scaling, trend estimation etc. The user interaction is then to choose among alternatives and decide on some parameter values.

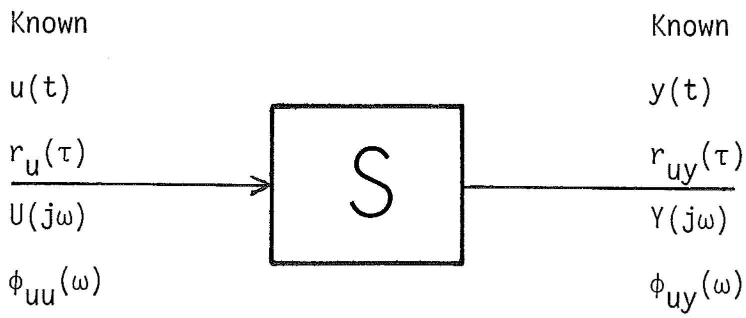
A more difficult problem is with more special purpose facilities such as linearization and the computation of more exotic statistical properties. The problem is that they will mean the inclusion of new code to be performed by the computer, rather than a choice between parameterized alternatives. This is a situation one order of magnitude more difficult to handle, as will be discussed in Section 3.1 'Computation structure'. The ability to interactively specify new computations is, however, very useful, as it solves the problem of how to provide facilities wich are indispensable for some users and not of interest to others.

Finally, let us observe that although this preliminary phase is typical for the case where operator interaction is valuable, cases are likely to occur where a large number of experiments are to be subject to the same treatment, i.e. a possibility to automate the operations would be nice. This is discussed in Section 3.1 'Interaction structure'.

2.3 Data Analysis - Non-parametric

This section serves to describe some standard methods used to find the dynamic properties of systems or signals, and the type of interaction likely to be used in applying them.

Dynamic properties of signals are often described by their covariance functions or their spectral densities. The spectral density can be computed from the covariance function or directly using a fourier transform technique. What happens to these properties when a signal passes through a dynamic system can be found theoretically, e.g. assuming stationary signals and linear systems. This gives some possibilities to determine the properties of the systems, once input-output signals from the system or some of their properties are known, see Figure 2.2.



Known relations:

$$y(t) = \int_0^\infty h(\tau)\ u(t-\tau)\ d\tau$$

$$r_{uy}(t) = \int_0^\infty h(\tau)\ r_u(t-\tau)\ d\tau$$

$$Y(j\omega) = H(j\omega)\ U(j\omega)$$

$$\phi_{uy}(\omega) = H(-j\omega)\ \phi_u(\omega)$$

Figure 2.2. Alternative known system input/output quantities and their known relations, usable in order to extract a description of a system.

Algorithms to perform the indicated computations are well-known; the Fast Fourier Transform to find the U(jw) and Y(jw), numeric deconvolution to find h(t), and the computation of $\Phi_{uy}(w)$ or $\Phi_{uu}(w)$ from $r_{uy}(\tau)$ and $r_{uu}(\tau)$. In practice, however, some problems occur, due to the influence of noise. Elimination of noise is e.g. done by averaging between different runs in the case of fourier transformation or by the use of windowing in the case of spectrum calculation. These techniques are often an area for operator intervention since parameters such as window width should be selected with regard to the noise level and the actual results obtained.

Interaction

Summarizing, we find that the needs for interaction in the task of data analysis are moderate. The basic need is for the operator to be able to specify which data to use and then to select a suitable algorithm. The only remaining need and where some experience and judgement capability can play a role is in the choice of the noise reduction parameter, i.e. the window width.

2.4 Identification

In this section we treat the computation of parametric models from measurements of the inputs and outputs of the system . A parametric model is e.g. a state space model or a rational transfer function as opposed to e.g. a transfer function given as a table in w of amplitude and phase. The importance of parametric models is due to the fact that a number of design methods require this description of the given system.

Let us first assume that the system indeed is given by a tabulated transfer function, either as a result of a direct frequency analysis experiment, or as a result of operations described in the previous section. The adaption of a rational transfer function to exhibit roughly the same frequency response would be one way of obtaining a parametric model. This operation has been investigated in the literature [Bos172] using various numeric methods to minimize the difference between the given frequency response and the one computed from the model. Problems have sometimes been encountered. Several of the difficulties are typical of situations where a human operator would be able to play a role, e.g. to determine a desired model parameterization; order, real or complex poles/zeros etc. The operator would also be able to indicate which part of a given amplitude/phase characteristic is most important to imitate.

Identification means the computation of parameters in a model of some assumed structure from a registration of the input and output of the system as functions of time. Examples of model structures used are transfer functions, multi input ARMA models and state space representations. A noise input is often included. Depending on the assumptions made on the disturbances and on the structure of the model, several different methods are known from literature. Usually, the methods give results with known statistical properties. This gives a possibility to answer questions on the selection of a proper model order, either through a test on the achieved loss function reduction, or through tests on the residuals, i.e. the estimated white noise in the noise model.

There is not yet any totally satisfactory way of choosing model order and other structural indices. Hence, this operation can not be automated, rather this is an area where a human mind still can contribute. This is so because the desision must be based not only on the value of test quantities, but also on a-priori knowledge of the identified system and qualitative judgements of the properties of the achieved model. The final choice will also be influenced by the intended use of the model. The desired properties of a model to be used for control design will depend on the design method and will be different from those of a model used to gain detailed understanding of the process itself. Cf. Example 1.

Other things may come up, calling for human aid. One example is to guarantee that convergence to a global extremal point has been achieved. This is typically done by altering the starting point in the cases where a hill-climbing algorithm is part of the identification method. Finally, some conditions may arise where a decision to revert to an earlier stage might be necessary, maybe as early in the path as a new experiment. A few examples illustrate this.

a) A previously undetected measurement error is likely to show in the residuals. Go back to the preliminary analysis phase and try to correct the erroneous point.
b) If the results are not as expected, one possible cause is that the measurements contain a bias. Go back to the preliminary analysis phase and try to subtract mean values or remove trends.
c) The accuracy in the parameters are poor. This is probably due to the measured input-output sequence being too short or containing too little power in the important frequency range. Redesign the experiment and go back and make a new one.

Methods exist for identification of models of many different forms. The problems and needs for interaction described above are always relevant, but in some cases there are new ones. A demanding problem and one where interaction would be of great importance is in the identification of non-linear models. Known practical techniques allow determination of parameters in such a model, but the specification of the model structure, i.e. the form of the non-linear differential equations, has to be done through human intervention. The problems arising in interactive specification of equations are discussed in Section 3.1 'Computation structure'.

Interaction

The interaction needed in the identification field is of several types. Of course, there are the basic needs, the specification of method (and desired model order) and of which input-output data to be used. Then there may be a second level where initial estimates are defined or parameters are fixed to certain values, additional outputs are demanded etc. After the identification the results obtained will be scrutinized by various methods, including visual inspection. The operator will probably then change the set of parameters governing the identification algorithm in order to improve the result. Thus, the identification process is an iterative one, with heavy numeric computations interspersed with operator actions of many different kinds.

In the case of a series of similar measurements on the same object, it is reasonable to expect that a certain procedure will evolve that will solve the problems intrinsic to that object and its environment. For that particular application it would probably be desirable to be able to define such a procedure once and for all and be able to invoke it easily and also to be able to easily make minor modifications.

2.5 Model Building

In the preceeding two sections we encountered methods to obtain a system model by mathematical operations on measured data, assuming that the system already existed. This may not be the case, or measurements may not be feasible because they would violate safety requirements or impair production. In this situation, a model could be obtained through application of general knowledge of physics, chemistry, mechanics etc. Modelbuilding on this basis may show difficulties, maybe severe ones, not generally in the area of computing and datahandling, but rather in assessing what are essential properties and relations of the system. This is a task for the model designer and no computer can do this for him. However, some sub-problems can be handled.

A good simulation facility, see later and [Elmq75], may be helpful. It would give a possibility to check the credibility of partial results during the various stages of development. Likewise, an advanced facility like the one in [Elmq78] will offer services like generic submodels, a powerful connection operation and the use of implicit relationships.

A risk in model building is to include some aspects in much detail just because the details are known. A good simulation facility can help to determine those details that contribute to the overall credibility of the model.

The modelling effort might give a result in the form of a system of non-linear first order differential equations. Although very valuable for the assessment of the final design and for other purposes, such a model would have to be linearized before it could be used in most design schemes. This could be done either by formula manipulation methods, or through numeric differentiation. Although techniques for formula manipulation are known and have been so for many years, no application of this type is known. In general, such methods seem to imply more severe demands on the programming language than are otherwise common in interactive programs. Linearization by numeric methods would not be too difficult to include in an interactive non-linear simulation program. The additional requirements on interaction would probably be limited to the specification of an operating point and one or two algorithm parameters.

A model of a system may also be obtained from another and more detailed one. A high order model might be required to describe all aspects of a given system but could be unnecessarily complex for control design. Techniques are available that tries to retain essential dynamic properties of a model but with lower order. The methods are all somewhat arbitrary, so human interaction is certainly advantageous. Criteria such as step- or impulse-response or frequency response agreement would have to be judged. The performance can be improved by altering weighting factors, the order of the reduced model etc.

Interaction

Summarizing, the field of interactive model building is to a large extent unexplored. Requirements are mostly basic - choosing method and algorith parameters. In model order reduction, elements as result judgement and iteration are found.

2.6 Analysis

The analysis phase serves to gain knowledge of the properties of the system model. The type of questions asked are e.g.:

- Which input influences which output?
- What is the bandwidth?
- What is the degree of stability?
- Is the stepresponse oscillatory?
- Is the system observable (controllable)?
- Is there a steady state error in the control loop?

Such questions could be asked during the model building phase. Some are relevant only for linear systems, some can be given a quantitative answer, others are more qualitative in nature. Answers to questions like the ones above will indicate if the model is sound, or if something is wrong. Likewise, during system construction, questions about observability and controllability will give answers on what transducers and actuators have to be included in a new system. In control system design, answers will indicate what method to use, what design results and performance to expect and during the work, will help to evaluate the achievements.

In general, many control system design methods are based on certain analysis techniques. This will be elaborated on in Section 2.8. As have been hinted at above, analysis methods are bound to play a significant role in modelling and system construction as well. The analysis methods are thus likely to be used iteratively, bringing interaction into focus.

Let us first observe that simulation may in many cases serve as a powerful tool for analysis. The trained human operator will draw many useful conclusions from the curves obtained. Although the information may be in a less quantitative form, it is likely to have a strong intuitive appeal. Simulation is treated at greater length in Section 2.9.

It is important to note in the following that some quantities of interest may be found by inspection if the system is on a special form, while if not, quite complicated computations might be needed. Thus there is a strong connection to the next section, Transformations. The system order is such a quantity. In many cases (e.g. state space form) it is apparent directly. For a transfer function matrix, however, the computation is much more difficult.

The question of controllability/observability can be answered for systems on state-space or polynomial matrix form. With some additional effort, a decomposition in state space of controllable and observable, uncontrollable and observable etc. etc. subsystems can be obtained. The practical problem of assessing "how controllable" etc. is more difficult, but can be solved.

The question of stability is similar in that methods exist, and are used, that give the same type of YES/NO answers. However, many analysis tools such as root locus computation, eigenvalue - pole computation etc. not only give a direct answer concerning stability, they also give an indication of the degree of stability. These methods are also often the basis for design. Frequency responses often convey the same type of information, i.e. stability and degree of stability can be determined. With some experience, qualitative properties of the system responses can also be extracted. Furthermore, the computation of the frequency response of systems on transfer function or state equation form is rather straightforward, and the display of the result in the form of Bode, Nichols & Nyquist diagrams have strong traditions.

Finally, it should be emphasized that quite a number of quantities like bandwidth, step response, rise time, solution time, error & stiffness coefficients, etc. in most cases are easy to compute. The inclusion of such facilities is more or less a matter of taste.

Interaction

The needs on the interaction are mainly the basic ones: choice of operation and specification of the system and representation in question. Operator intervention in an analysis method as such is not generally needed. However, the analysis operations may be intimately connected with other (iterative) operations such as design, and it is quite possible that it would be natural to group some steps together into a sequence to solve common subproblems.

2.7 Transformations

A system may be represented in many different forms, see Section 5.3. Transformations between different forms exist. Generally speaking, no special needs for interaction are present, mainly because of the nature of the problem. For a transformation to be interesting, it should be one-to-one and independent of explicit choices.

There are two exceptions to this, one is the transformation from a polynomial matrix form to state space form. The selection of state variables may not be unique but could be done interactively to ensure that the most natural choices be made.

The other one is where several subsystems are combined into a single transformed system. The specification of the desired connections would differ according to which quantity of the combined system is considered interesting, e.g. the control signal or an internal variable in a closed loop system. Interesting signals should be included as outputs of the system. An example is the command SYSOP used frequently in Example 2.

2.8 Synthesis

This section will discuss synthesis of linear automatic control systems. In this treatise this problem is considered the main one, the earlier sections and the following ones describe subproblems encountered. Therefore this section will be somewhat longer. The first subsection will discuss general ideas on control system design. The next will serve to classify the problem and to treat some design methods. Frequency domain methods and time domain methods are then treated in two separate subsections. Finally an attempt to condense the interaction requirements closes this section.

General Ideas

The synthesis procedure aims at obtaining a certain goal. Sometimes, this goal can be expressed as a set of relations to be satisfied. Examples are requirements on bandwidth, error coefficients, amplitude- and phase margin, solution time, rise time, overshoot etc. etc. These quantities represent knowledge of desirable and/or attainable characteristics in terms of precision, speed, and stability. It should also be noted that these quantities often describe the same thing. The rise time measures the speed of response, as does the bandwidth; the amplitude and phase margin measures stability, as does overshoot and solution time. The type of specifications used are a function of the intended use of the system, the design method and the designer's intuitive feeling, experience, and preferences. In some cases, aspects of the design goal are hard or impossible to define in terms of numbers. Instead, the designer has to use his own notion of what "nice" properties might imply. An example is the design of aeroplane dynamics, where subjective criteria (pilot rating) play an important role.

In many cases, synthesis consists of extending methods and theory to a new field where they approximate the real problem. An obvious example is the use of linear methods almost everywhere although the real life in most cases is non-linear. The art in synthesis, as in engineering in general, is to choose the appropriate method of approximation, to apply common sense, intuition, and a-priori knowledge. Some ways of doing this has proven sufficiently successful to be referred to as synthesis methods. They are a combination of some analysis method, maybe in a slightly modified form, together with special rules and concepts that help in altering some defined design parameters in such a way that the resulting change in the analysis is predictable and the desired one.

It is clear from the above discussion that synthesis methods are with few exceptions iterative, and hence, if used on a computer, interaction is essential. In the following, we will try to show that this general characterization fits on some of the commonly used methods.

Classification of the problem

Figure 2.3 is an attempt at a classification of design problems and methods. Generalizations are dangerous if they are treated as the ultimate truth, but it is hoped that this figure may convey some useful notions.

Starting at the left hand side, we see that if it is possible to use a high gain, this is correlated to the possibility to solve problems with low a-priori knowledge, simply because then a tight feedback loop is feasible. On the other hand, if available gain is low, one is forced to try to learn as much as possible about the dynamics of the system and its disturbances. The available gain is determined by practical considerations such as the amount of

Available Gain	A-priori knowledge	System Description	Methods
	High	State Space	(Stochastic) observers
Low	Internal	High order	Linear quadratic
	Detailed	Differential equations	Pole assignment
	Low	Transfer function	Root locus Frequency response
High	External	Impulse response	Transfer function specification
	Input-Output		

Figure 2.3 An attempt at a general classification of design problems

power or energy possible to put into the system, the size of actuators etc. Another limiting factor is the noise level in the system, i.e. a constraint on the information side.

The next two columns correlate the degree of knowledge with the type of system description used. High knowledge is taken as meaning a detailed model describing the internal structure of the system, i.e. we know what is going on inside. System representation forms that utilize this type of knowledge is of course a state space representation or a system of high order differential equations. In contrast to these two we have transfer function or impulse response representations, which forms only describe the dependence between inputs and outputs of the system. What is going on inside is not known, and, which is important, can not be utilized by any design method.

Of course, knowledge means effort and money. Therefore, methods with less demands in this respect have a great importance. On the other hand, little knowledge means that it is hard to judge achievements in relation to what might have been attainable, i.e. what is good and what is bad?

Some Synthesis Methods

Some methods of synthesis will be described here. First we treat transfer function methods. The main method, or rather class of methods, are frequency response methods, described in a separate subsection. Other methods, treated here, are the root-locus method and the "transfer function specification" method. Finally, one method based on state space system descriptions is mentioned, namely pole assignment.

Root locus

The root-locus method is a good example of the situation where the computational powers of the computer can interactively serve the control system designer. The method, primarily applicable to single input - single output systems, is basically a stability analysis method. The important and valuable feature is that the degree of stability as a function of the varying parameter is assessible from a graph produced by the computer program. Thus a method of studying the influence of varying parameters is offered. Any parameter could be varied, but the case with varying loop gain is the most important and here, simple rules of modifying the regulator to achieve desired improvements exist. The root locus method has recently been extended to the multivariable case, [MFar77].

Needs on interaction are: good means of obtaining a nice graphical output, specifying which is the parameter of interest, what range of values it may take, and what is the structure of the system (it might be composed of subsystems).

Transfer function specification

The formulation of the synthesis problem simply as an equation: "Given a system so and so, compute a regulator so that the closed loop system becomes ....", might be the first thought of the novice. Indeed, the problem could be solved that way, only that there are some pitfalls to be avoided. Here too, the computer can offer some help. It is natural to assign the numerical solution of the equations to a computer. Care must be exercised to make a reasonable specification, i.e. results from the analysis of the given system must be incorporated. The solution might not be unique, so some choices would have to be made, and their consequences analysed.

Again, interaction represents the important factor that makes this method practically useful. By structuring the computations to allow suitable interaction points, existing analysis methods and parameter alteration operations can be brought to bear on the difficulties above.

Pole assignment.

Pole assignment methods are similar to the "transfer function specification" methods in that the system is given together with a desired dynamical behaviour. Also the solution is similar; it consists of a computational part, suitable for computer implementation and a specification part. The specification part contains an analysis of what is reasonable to demand, and a choice (in the general case) of a legal feedback structure. The needs on interaction are as before.

Frequency domain methods

These methods depend on a stability analysis result formulated as restrictions on the graph of G(s) for certain values of s. Design aims at through proper choice of regulator parameters and regulator structure make the graph "behave nicely". Although objective definitions of "nice" do exist - amplitude and phase margin, the precise interpretation from shape of curves to that of corresponding time responses is left to the designer and his experience.

In the classical SISO problem, the curves can be represented in different forms - Bode, Nichols and Nyquist diagrams, familiar to a generation of engineers. Recent developments [Rose69] and [MFar73], tries to draw on this familiarity of thinking for the MIMO case. In addition to the shaping of individual curves, problems arise from the multivariate nature of the system. In the INA method (Rosenbrock) analysis tools in the form of e.g. Gershgorin circles are integrated into the design scheme. They are used to help in making the system diagonally dominant, whereafter reasoning familiar to the SISO case will apply. The numerical computations involved are the evaluation of a matrix of rational transfer functions for different values of s=jw and computing the inverse of the complex matrices obtained. The characteristic loci method (MacFarlane) can be thought of as making Nyquist plots of the eigenvalues of the transfer function matrix, again evaluated for s=jw. The eigenvalue plots together with plots of the angles between eigenvectors are used to determine and influence the degree of stability and interaction in the system. Recently it has been suggested to use singular values rather than eigenvalues [Doyl79].

Unlike the SISO case where graphical methods with pen and paper is able to solve the problems, the MIMO methods relys for their practical usefulness on interactive computer programs. The effort needed to draw the graphs by hand would be formidable. The needs on the interaction facility includes functions to guide in the display of curves on a graphical computer output, as well as for help in designing compensation dynamics. The order in which these operations are called upon is not predictible and some of them, e.g. editing graphical output and connecting elementary controllers together, may be available in a more general framework. Again the need for a basically unstructured but structurable interaction tool arises.

Some paths in the design, for instance simulation of the closed loop system in its current stage of development, will use many general purpose facilities. Examples are forming the closed loop system from its subsystems, transformation to a form suitable for simulation, the actual simulation and the display of the resulting time-responses.

Time domain methods

Although the historically oldest method, design in the time domain became feasible for practical problems in the sixties when computers began to be available as tools in engineering. The methods are characterized by the type of model used - systems of 1st order linear differential equations. Such an internal description not only describes how outputs are dependent on inputs, but also the interdependence of internal variables, often state variables.

Results from synthesis are typically in the form of a linear feedback from all state variables, giving a rich class of controls. The most important design method is based on what is known as linear quadratic control theory, giving a result in the form mentioned above. Analysis/evaluation of the achieved results could be by a number of methods, but most importantly through simulation of transient responses. Specifications in the time domain are then easily checked. Changes in the performance is accomplished by altering weighting factors in a quadratic criterion, a procedure which is intuitively simple and easy to learn.

It is important that the behaviour of internal variables and control signals is available for study. Also, physical knowledge can be used to judge whether the performance is reasonable or not.

In practice, all state variables are not available for feedback, and if they are, their measured values might not be suited for direct use due to noise corruption. One solution is to include dynamics in the feedback through state observers. They could be of different kinds, Kalman filters, Luenberger observers etc. Encouragingly enough, they can be designed by the same algorithms and with similar interaction as used for feedback design. An important feature with feedback from reconstructured states is that measurements need not be the same as the controlled variables.

As before, the design is iterative and interaction is most important. Some operations will be specific for this design method, but many means of analysing the current design stage will be general purpose in nature. Examples are eigenvalue computation, composition of a closed loop system from subsystems, simulation, and plotting of time responses.

Interaction

Synthesis methods use tools in analysis for specialized purposes. In many cases though, general purpose analysis tools can be used, if adequately designed. A synthesis method often includes a planned form for the change of design parameters. Again, there is a competition between generality and specialization. Here interactive facilities may play a role. It might be possible that the system for interaction allows an adaptation of the interaction to different needs. An example is given in Section 4.6.

Most important, however, is that synthesis methods require an interactive approach. In the iteration loop, operations like editing output in graphic form, changing design parameters, analysing results and computing an improved design will be found. The details of this sequence are likely to vary from case to case, but they may also be unchanged within the solution of a specific case, or class of cases. Thus (local) standard procedures will develop. It is of prime interest that an interaction scheme will allow this to happen. This is discussed in Section 3.1 'Interaction structure' and is exemplified in Section 4.6 and in Example 2.

2.9 Simulation

Simulation is a problem area in its own right, treated e.g. in [Elmq75] and [Elmq78]. Aspects relevant in this context are the uses, needs, and demands on interaction as a tool in analysis and synthesis of control systems.

Simulation as a tool is important in three different functions:

a) As analysis tool to learn things of the system, viz. the dynamical behaviour of the system in a wide sense: signal paths, time constants, type of response etc.
b) As analysis tool in synthesis. To inspect how the dynamic behaviour is changed by the proposed design and evaluate how well specifications in the time domain are fulfilled.
c) As a cheap, safe and easy-to-use testing tool of the design on a maybe more complex (non-linear) model prior to the implementation.

It is important to note that in case b) (synthesis) and many times in case a), the model is linear. This makes the simulation problem simple, the model having a known structure, easily and efficiently implementable in a computer program. In the case of a non-linear model, however, as might be used in cases c) and a), not only the parameter values but also the structure of the model must be specified. To do this interactively poses a much greater problem, cf. Section 3.1 'Computation structure'.

Interaction

When simulation is used in design, it is in principle used as a transformation, viz. from a system represented in e.g. state space form to a representation as a pair of inputs and outputs. The analysis is performed by eye after the time-series have been visualized. The interaction needed restricts itself to the choice of operation, of input and output and of the system in question.

In the more general simulation problems a) and c), interaction plays an extremely important role. The possibility to alter parameters and maybe change the structure of the system while rapidly being able to study the corresponding behaviour of the system for different inputs and/or initial conditions, is a most powerful way of examining the system and of gaining an intuitive feeling for its behaviour. Here the demands on interaction facilities are high indeed. Not only must the means of freely accessing the model be flexible and easy to use, the means of displaying results must also be well developed. In this kind of application, the freedom available to the user is of major concern, as is the possibility to automate some of the interaction. Examples are the running of Monte Carlo simulations and batch simulations (sic!). Cf. the discussion in Section 3.9.

2.10. Implementation

This chapter has described in varying degrees of detail the steps taken in the design of a control system. We started with the aquiring of measurement data to determine parameter values, and the natural last step would be the implementation phase. Can that be supported by interactive software?

Yes, in some cases this would be natural. A company which either makes control hardware or is a big user of such hardware is likely to design many control loops where the hardware to be used in the implementation is known. In such a case, it would be reasonable to include in the software a facility, to output the result of a design process in a form suitable for direct transfer to the intended hardware. A special and very interesting case would be when the implementing hardware is a computer, either the same as the one running the design software or connected with it. Then the transfer of the resulting regulator parameters could take place without human intervention. The facility sketched here is however not known to have been tested anywhere, and there is no experience to report.

3. INTERACTION PHILOSOPHIES, PROGRAM ORGANIZATION AND THE INTERACTIVE USER.

This chapter will begin by examining the different desirable interaction forms found in the previous chapter. This is done in Section 1 while Section 2 tries to formalize these results, also taking into account the general need for guidance and information and the correction of errors.

The two main forms of organizing an interactive program, question & answer dialogue and command dialogue, are treated in Section 3 to 6. Their intrinsic properties, possibilities and needs are discussed. Section 3 treats the question & answer type dialogue while Section 5 treats the command type. Sections 4 and 6 have an identical structure, comparing the two dialogue types property for property.

In Section 7, we review the demands that stem from program organization considerations. How these demands are satisfied by programs constructed according to the two interaction strategies are then discussed in Section 8.

Section 9 tries to characterize the interactive user and demonstrates that he may take several different shapes. Section 10 finally presents the proposed policy in interactive program design.

3.1 Interaction Needs

We are now going to discuss the needs we have on interaction. The interest will be focused on the type of information that has to be exchanged between the problem solver and the computer. The form of the interaction will be discussed later. The previous chapter will serve as reference and motivation. The four typical needs are called:

(1) Choices and parameters
(2) Multi level interaction
(3) Computation structure
(4) Interaction structure

Note that the interaction needs listed here are typical not only to automatic control problem solving, but are general to a wide class of situations where a computer is used to help a designer with heavy computations or data handling operations. Typical are also the existence of well structured data objects (Chapter 5) and the element of human intervention in the operations.

There certainly are disciplines with other interaction needs, e.g. circuit design, computer draughting and inventory control.

(1) Choices and parameters

The most common and most basic information the user of an interactive program will have to pass is what choices he has made and what parameters he wants to use. The choices to be made include the action to be performed and the datasets which are inputs and outputs of a given program module. Parameter values must also be specified. They represent a possibility to influence the operation in a predefined way.

In this type of interaction, the operation is fixed and given by the program code, and the input and parameters are the only freedom left for the user. This situation is by far the most common one. It is typically found in analysis and specification type operations. Hence it also plays an important role in the synthesis and measurement situations.

(2) Multi-level interaction

With multi-level interaction is simply meant that the interaction is split into two or more levels. This situation occurs when proper parameter values, appropriate secondary input or other choices in the applied method are not apparent until some preliminary computations have been performed. It is often possible and most attractive to divide such operations into two or several parts, allowing common analysis tools to be used to determine suitable future steps. If, however, the information to pass between the different parts is special in structure or the analysis needed is not of a general nature, it would be more natural to implement the method in a single program module but allow interaction in several levels. This would also be the case where a number of options exist. If they had to be specified all at the same time, it would be clumsy, difficult to comprehend and remember and would be generally unaestetic.

The solution is to allow interaction in several levels. The first level is used to specify the problem. On the next level the problem is analysed, or details of its solution are entered. In the general case, temporary results could be asked for and allowed to influence the user's actions on the lower levels.

Examples were found in the section on Identification (2.4). One applies to the fitting of a transfer function to a frequency response, where the second level of interaction treats details of the curve fitting method. The second example was found in the maximum likelihood identification method, where the lower level of interaction is used to optionally specify the starting point, values of fixed parameters etc.

(3) Computation structure

A need was found in the general data analysis operation, in identification of non-linear models and primarily in the simulation of non-linear systems, to be able to interactively specify a series of computations. Unlike the previous needs where the computations are fixed and only data sets and and parameters are changed, we here encounter a need to specify the series of operations themselves. In a computer system, this is a task solved by what is called compilers or interpreters, usually large and expensive programs.

The difference with this interaction need compared to the previous ones is that it involves to parse statements in some arithmetic language, obeying its syntactical rules, and to generate a sequence of (computer) instructions that performs the intended task. It is indeed possible to include such facilities in an interactive program, see references [Elmq75] and [Hall78]. In this report, however, we will not explore this need any further, apart from the notes on hypothetical future programming languages in Sections 5.5 and 6.6.

(4) Interaction structure

In many places in Chapter 2, it was noted that there was a desire of being able to specify an interaction structure, see e.g. Sections 2.1 and 2.8 (Measurements and Synthesis). Such a facility would be useful either for temporary or more permanent use.

By interaction structure is understood a fixed sequence of interactions that can be invoked easily, maybe with some planned alterations. The temporary use of such a sequence would be very natural for the interactive user that solves a problem with a partly iterative method.

The more permanent interaction sequence serves to build new functions from other more basic ones. This could be used to implement methods applicable to certain problems, e.g. a synthesis method, or to construct interaction modules aimed at a certain cathegory of users, e.g. students. See more details in Section 4.7.

Note that this so called 'interaction structure' bears strong resemblance to the earlier 'computation structure'. To a degree, the same desired result could be achieved by this facility, provided suitable basic functions were available and called in proper order.

In fact, this is the key difference between the two concepts. Here, talking of 'interaction structure', a very simple syntax is assumed. The elementary operations in the interactive program are called one at a time, with proper operands. The only rules to obey is in the choosing of operands, something that in any case must be checked, presumably in the code implementing the elementary operations.

3.2 Interaction Models

In the following sections we will try to formalize some interaction types satisfying the needs listed in the previous section. Their properties will be listed with some discussion. In addition to the needs that have been noted so far, coming from a typical application field, we will also include needs of a practical nature, such as possibility of error correction and acquisition of guidance and help. The interaction types will be described using state diagrams, as has been done in [Aaro77].

The states are indicated with circles with numbers, connected by lines marking possible state transitions. A state represents an interaction point, i.e. a point in the program where the user must respond to a result or a question output on the terminal. His answer may influence the future state transitions, i.e. the way to follow along the lines as in Figure 3.1. Note that the computer calculations are done between interaction points, i.e. along the state transition lines, but the type and amount of work is not indicated.

The two dialogue types discussed, question & answer dialogue and command dialogue, are both directed towards the situation assumed in this dissertation; interaction through a computer terminal with keyboard and most often a graphical output. Reference [Aaro77] also treats what is called Mixed Dialogue and Escape Dialogue, applicable in this context. Mixed dialogue is a trivial combination of the main types. It has some nice properties in special situations and is treated as an example in Section 4.7. The second one is a way of alleviating the key problem in the question & answer dialogue, how to escape irrelevant questions when it is apparent that the current operation should be aborted. An example of how this facility could be employed is also found in Section 4.7.

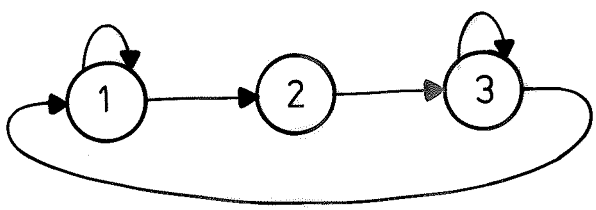


Figure 3.1 The type of state diagrams we will use.

Using a graphic output device with light-pen facility enables completely different forms of interaction. Section 6.3 describes some features possible and why these have not been further explored in this dissertation.

3.3 Question & Answer Dialogue

This type of dialogue is characterized by its large number of interaction states, see Figure 3.2. State 1 represents the situation where the user has received the question "what next?", and should answer giving his choice. One of several paths will be followed depending on the answer. The one marked ? represents the possibility of requesting a list of alternatives.

Each alternative the user may choose corresponds to an interaction loop such as 11-12-13 or 21-22. Let us assume that these states are used to learn the user's desires on input data, parameter values, options and output data. If

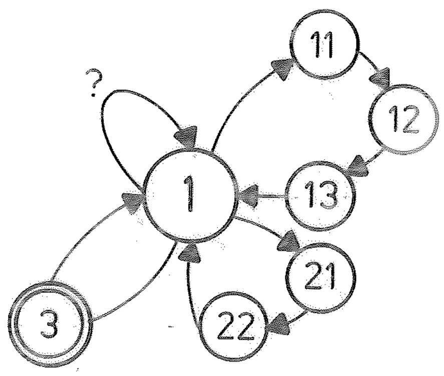


Figure 3.2 State diagram for a simple question & answer example.

so, we have thus obtained the facilities asked for in the paragraphs on 'Choices and parameters' in Section 3.1.

The compound state 3 serves as a more complicated example. Suppose it has the detailed state diagram shown in Figure 3.3. Interaction point 33 offers a choice of three alternatives. The first one doubles back into state 33 while the second goes directly to 34. The third reaches 34 first after additional interaction including another fork on the path in 3322.

Now assume states 31 and 32 serves to gain knowledge of the problem and its parameters. The path 3311 - 3312 might offer means of analyzing the problem while the two paths to 34 may represent different ways of its solution. We thus find that what we called multi-level interaction is easily included in the question & answer approach.

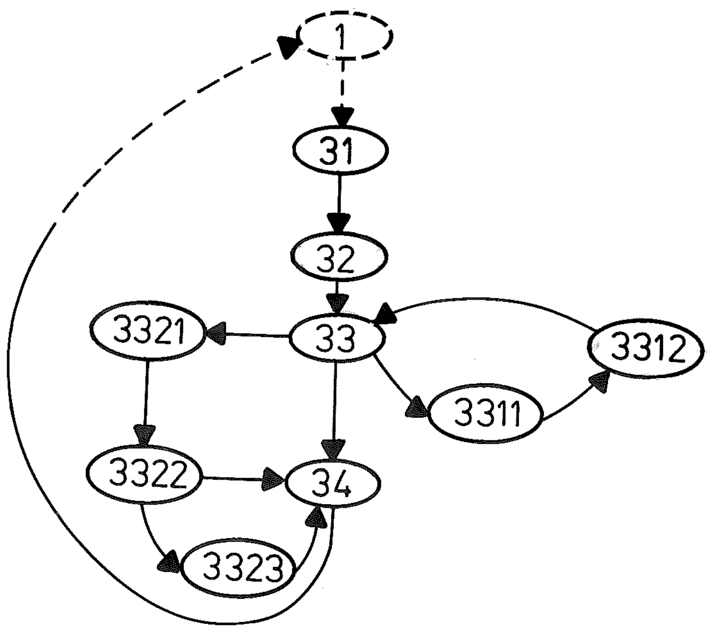


Figure 3.3 State diagram for the second level of interaction in Figure 3.2.

3.4 Properties of the Question & Answer Dialogue

The most important properties of the question & answer approach will be listed, together with some comments. They should be read in the context of Section 3.1. A corresponding list for the command dialogue is given in Section 3.6.

a) Meets demands (1) & (2) in Section 3.1 easily.
   As demonstrated, state diagrams for question & answer interaction allowing choices and parameters to be passed to the program can easily be constructed, also in a multi-level form.

b) Can be implemented using simple programming languages.
   The interaction represented by the states shown in Figures 3.2 and 3.3 consists of the displaying of text, often in the form of a question, and of reading the user's response. This can readily be done in simple programming languages such as FORTRAN.

c) Defaults, typing error immunity.
   The use of so called defaults is often desirable. By this is understood the practise of showing the current or initial values of the requested variable and allowing the user to retain this value, e.g. by simply typing an empty line on his terminal. This is usually not possible within the framework of standard I/O in common programming languages. Typing error immunity implies that a simple typing error, such as including an alphabetic in a number, should not cause a serious run-time error. This is often the case though, and these two desires usually forces that input routines other than the standard ones be used in an interactive program. Cf. b) and Section 6.5.

d) The questions give good guidance.
The output to the user at the many interaction points may be formulated so that also the unaccustomed user will know what type of answer is expected of him. Likewise his answers will be tested for legality immediately.

e) Interaction fixed by the programmer.
As all interaction points are implemented in the program code, the programmer has a very responsible task. All possible tricks, options, or variations the user may want to use must be anticipated. Conversely, the user will only be able to perform precisely those functions the programmer has planned.

f) Large volume interaction, boring to the experienced user.
Many and detailed questions will be boring to the experienced and frequent user. Naturally, alternate forms of the questions could be provided, but this adds extra complexity to the program. Also, the question could be inhibited if the user has anticipated it and already has provided the answer. Again, this 'type ahead' facility costs extra complexity and specially designed input routines.

g) Fixed interaction, frustrating when a mistake has been made.
Quite frequently, it happens that a user finds that he has made an error or he changes his mind too late, so that he finds himself locked into an interaction path without interest or meaning to him. Indeed, it might eventually output results destroying things he wants to retain. What is needed is some escape mechanism, allowing the abortion of an unfortunate interaction path. Again this raises demands on the input routines of the program as well as pre-planned paths out of program modules.

h) No concise description; lengthy log.
It is always good practise to plan the work at the computer terminal prior to the actual interactive session. What is needed then is a description of all possible interaction points together with their interdependence. Such a description will be rather lengthy, as will be the log of a session. Such a log is valuable to document the results obtained, but to be useful it must include not only results but also the user's response to the questions, and the questions themselves. The risk with many and detailed questions is that valuable data in the log may be drowned by a lot of transient information.

i) Interaction structure is not natural.
The key to this facility is to be able to store in advance the answers to questions to come. This could be done, using facilities mentioned in e) and g). However, the exploring of these possibilities would lead to an itemization of the interaction, and introduction of notions belonging to the next interaction form (command dialogue). Strictly speaking, the program would not be question & answer oriented any more.

j) In the program, interaction is mingled with computations.
Typical question & answer programs with state diagrams like those in Figures 3.2 and 3.3 will contain interaction code mingled with computation code. This is the major advantage with this type of interaction, viz. the close connection between the program and the user, cf. d). However, from a programming point of view, such a program structure should be avoided, cf. Section 3.8.

k) A time-sharing implementation should use few interaction points.
In a time-sharing environment, an interactive program will be swapped out of primary memory when either it has

used up a time quantum, or it awaits input from the user terminal. Thus it is advantageous, from efficiency reasons, if computations are separated from interaction and if interaction points are few. Also, the user will experience a certain response time at each interaction point. Many such points will lengthen the time needed to solve a given problem.

3.5 Command Dialogue

The distinguishing property of the command dialogue is the simplicity of its state diagram, see Figure 3.4. This figure describes the same interaction problem as Figures 3.2 and 3.3 combined. The basic interaction state 1 demands a new command from the user. In many cases as in commands C1 and C2, they are executed immediately and we return directly to state 1.

In a case like command C3, we enter a second level of interaction. Here a number of subcommands are available, similar but distinct from the commands on the main level. As in the analogous example in Figure 3.3, subcommand C21 might serve as a tool for analysis, while C22 and C23 are alternative ways of solving the problem, chosen with the help of the previous analysis.

The way the commands convey information on the user's choices and his desired parameter values is by means of arguments in the command. A simple command could have the form:

CMND ARG1 ARG2 (E1)

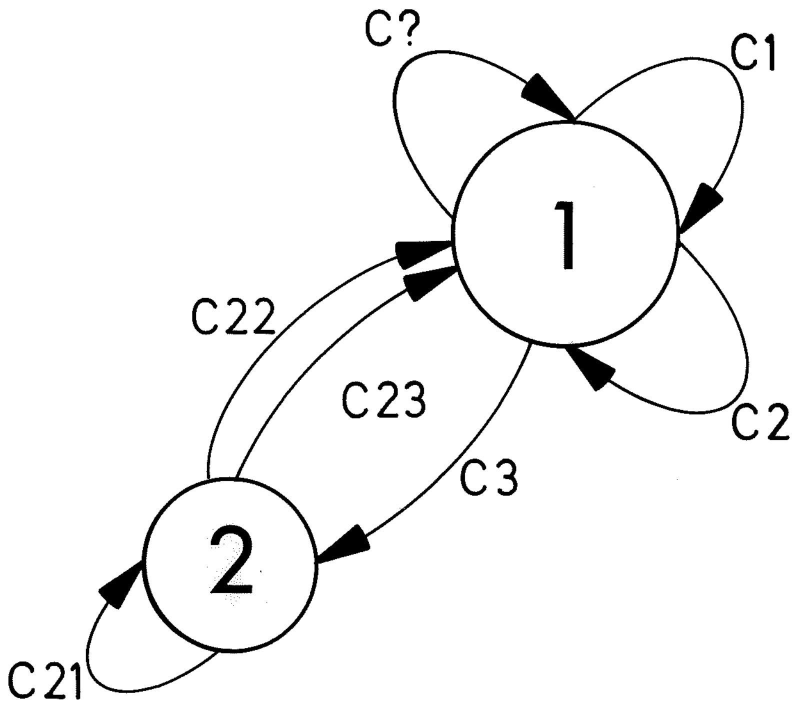


Figure 3.4 State diagram for a command dialogue equivalent to the actions in Figures 3.2 and 3.3.

This command line contains three information carrying items: the action required, and the two arguments. Thus this single line, that might be the generic form of C1 in Figure 3.4, could be equivalent to the state sequence 1-21-22-1 in Figure 3.2. The reason why the state diagram of the command dialogue is simple is that the command line contains more information than does a single answer in the question & answer dialogue.

By defining syntax rules also the form of the argument string can be used to bear information. Thus a generic command form

CMND [ARG34 =] ARG3321 ARG3322 [ARG3323] (E2) where [ ] indicates that the argument is optional, could be used to describe how to go from state 3 to state 1 in Figure 3.2. Note that the notation is chosen in accordance with that figure. The reason to include the equal sign is twofold. By separating arguments relevant to the output (ARG34) from the other, the argument string becomes easier to memorize. Also, the extra structure introduced allows an unambigous decoding of the command string, in this case when three arguments are used (Which one is missing? ARG34 or ARG3323? Look for the equal sign!).

3.6 Properties of the Command Dialogue

Again we will try to list the main properties of the dialogue form under discussion. An assessment of their relative importance and their relation to those of other dialogue forms will be made in a later section.

a) Meets demands (1) & (2) in Section 3.1 easily.
   Very simple state diagrams can be constructed meeting these needs. An example was given in Figure 3.4.

b) Specially designed decoding routines.
   The reading and decoding of commands like the examples E1 and E2 in the previous section is not possible using standard facilities in some programming languages in general use. Special application independent decoding routines can be constructed in all modern programming languages. Generally available FORTRAN implementations does not, however, allow computer independent coding of such routines, because of its lack of standardized character handling. This is further discussed in 6.5 b.

c) Syntax rules for the argument string.
The form of the argument string can be specified so that a number of objectives can be met. Arguments can be allowed to be optional and the inclusion of delimiters may make possible an unambiguous decoding with extensive checking of formal correctness. The extra structure introduced, grouping arguments together, will make the command look logically natural and therefore easy to memorize.

d) No guidance.
At all interaction points, the user is supposed to enter some command line, consisting of a command identifier (CMND in the examples above) and a string of arguments. The number of possible commands at any interaction point may be large and they may be of different forms. The novice is likely to feel abandoned. Of course some commands might be constructed to offer help by displaying a list of alternatives, command forms etc, but there is no step-by-step guidance.

e) Freedom from predefined interaction.
The proper sequence of actions to solve a problem is left entirely to the user. This relieves the programmer of the task of defining the intended use of the program. Rather, he should allow access to all possible parameters of the algorithm he is implementing, using optional arguments, optional subcommands etc. If there also are sufficient general purpose functions available, a user of the program might put the available facilities to a use the implementor might not have been aware of.

f) Free interaction; suitable to the experienced user.
The experienced user, i.e. the person with good knowledge of the underlying theory and who uses an interactive program regularly is likely to be prepared or indeed anxious to take the initiative in the communication with

the computer. In such a case, the command dialogue offers the possibility of a natural language in which the user can freely express his wishes.

g) Free interaction; good when a mistake has been made.
Mistakes do occur, either as a result of actual errors in thought or in typing, or in the form of a change of mind from the user's side. The free interaction where any command at a certain interaction point is equally possible, poses no objection to a jump back in the command sequence to a suitable point where to start again.

h) Concise description; concise log.
A command oriented interactive program can be thought of as implementing an interpreter for a specific problem solving language. The set of available commands are the statements in that language, and there are well established and applicable methods available to describe the syntax of the commands. Similarly, the log of the actions performed during a session will be short and easy to survey. The log is important as it shows the names given to the data and the way they were generated. Examples are given in the appendix.

i) Interaction structure is natural.
The close relationship between command dialogue and programming languages was mentioned above in h). It is very natural to carry across well-known principles such as procedures, I/O-statements and structural statements. The influence of such facilities on the possibilities of the command dialogue is explored in Section 4.7.

j) Programming structure; interaction isolated.
The form of the interaction state diagram for a command dialogue was shown in Figure 3.4. It is apparent that interaction, i.e. calls to command recieving and decoding

routines will be localized to a few points and that the computations are performed in a number of parallell paths void of interactions. In the case of subcommands, the same applies at a lower level.

k) Ideal for time-shared use.
By the arguments given in paragraph k in Section 3.4, greatest efficiency in an implementation on a time-shared computer is achieved for programs that seldom read from the terminal and that have interaction separated from computations. These rules are well satisfied.

3.7 Demands on Program Organization

In this section we will try to list some of the demands on interactive programs that arise from the programming point of view. Some are general with no specific influence on this type of programs and will not be discussed to any great length, others have already been encountered.

Portability

The portability of a program means its ability to be run on other computers of comparable or greater size but with other organization. There are some simple rules to follow in order to achieve a high degree of portability. First of all, the programming should be done in the standard dialect of a commonly used programming language. Secondly, parts of a program that have to be computer dependent should be confined in small separate program modules, so as to be easily identified and modifiable. Examples of such computer dependent parts are file I/O, non-standard I/O of textstrings, graphic (display) handling, and numeric test quantities.

Maintainability

Here we are interested in the possibilities to correct/modify portions of a program without affecting the rest of it. The solution lies in the proper structuring of program code, and not least, structuring and storage method of data. The last problem is discussed in Chapter 5.

A practical problem arises because programs tend to be large, consisting maybe of several hundred modules. A way out of this situation would be to split the program into several separate parts that, being smaller, would be easier to maintain. In the case of a command dialogue, the parts performing the computations would be a natural choice. The idea would then be to make the main part of the program call the other parts as separate programs when their services are needed. Unlike subroutine (procedure) calls, the existence and way of implementation of this facility is a function of the operating system on a specific computer. However nice, this solution thus violates the demands on portability.

Expandability

The ease of including a new facility may be of importance in many projects. There are a few factors that will promote this quality. One is the frequent use of primitives, i.e. common operations are made available as separate modules forming a pool of ready-made building blocks to glean from. Another one is that the data objects, the program is made to handle, are so structured that different portions of the program can be independent and be able to communicate through them only.

Segmentation

Interactive programs for general use will always be segmented on computers lacking some form of virtual memory system. The reason is either that the primary memory is too small or that there are restrictions on how much that may be used by any one user. The last situation applies to time-shared implementations. The ease with which such a segmentation can be made depends on the internal structure of the program.

Locality

On computers with virtual memory systems, programs need no segmentation, at least if the address space is sufficient. Instead there is a desire to have good locality in the program. This means that the points in address space referenced during a short period of time should be grouped together as well as possible. This will minimize the number of pages to be kept in primary memory as well as the number of page transfers from mass memory.

Modularity

There is a desire that the program code is divided into suitable modules, i.e. subroutines or procedures. Apart from being a result of good programming practice in general, this will be the key to the satisfying of the other demands above.

3.8 Effect of Dialogue Type on Program Organization

The difference in the general form of the state diagram for the question & answer dialogue and the command dialogue is reflected in the corresponding program flowcharts. Figure 3.5 shows a skeleton flowchart for a question & answer dialogue program, corresponding to Figures 3.2 and 3.3. QUANDA, short for QUestion AND Answer, is assumed to be the name of a general routine to output a question and await the answer. The common action Compute signifies that some operations are done on the basis of answers recieved up to that point. Of course, the computations will be performed by many different routines, and naturally only a few of the instances will signify substantial computational effort. The point with question & answer dialogue is, however, that questions indicate the state of computations carried along as far as possible, and that answers are tested dynamically. Therefore the distinguishing property is that computations and interaction are heavily intermixed.

In the command dialogue case, however, this is not so, cf. Figure 3.6. COMMAND is assumed to be the name of a routine that reads a command line and divides it into its different items. When a proper path has been selected, depending on the command recieved, the items in the command line are decoded, presumably by routines logically close to COMMAND.

To each of the possible commands there is a corresponding path in the flowchart. All such paths look the same and are parallel. Only in the presence of subcommands is the simple parallel structure broken, but only to reappear at the lower level.

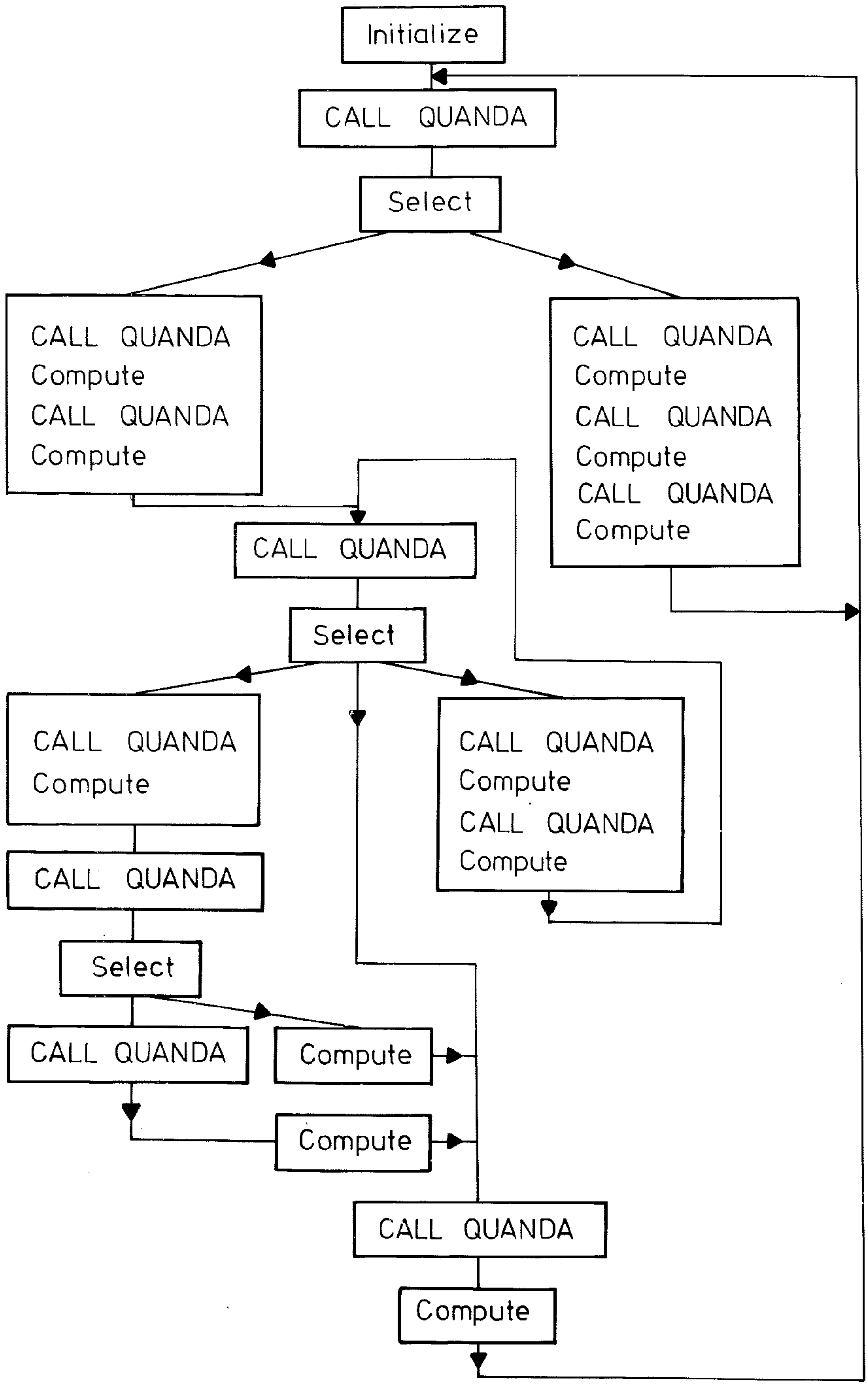


Figure 3.5 A skeleton flowchart for a question & answer program realizing the dialogue in Figures 3.2 and 3.3. (The path 21 - 22 is omitted).

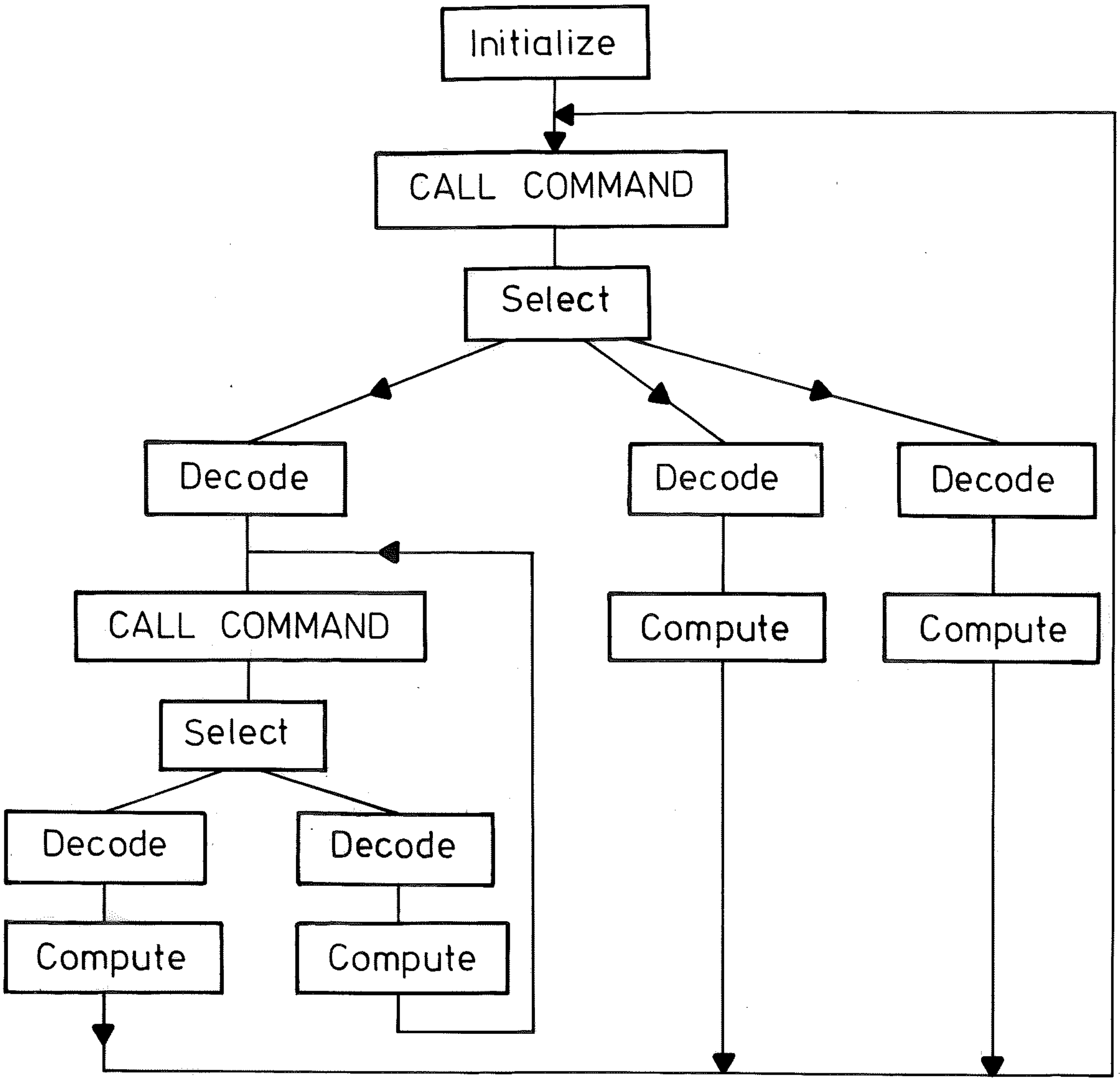


Figure 3.6 Skeleton flowchart for a command program realizing the dialogue in Figure 3.4.

Question & answer dialogue

The portability of this type of dialogue is positively affected by the basically simple form of interaction. Normally available I/O facilities will suffice, at least in principle. If more flexible input-output routines should be used, as discussed in Section 3.4c, they may very well have to be machine dependent.

Maintainability and expandability is largely dependent on the structuring of data. However, the program structure will also play a major role, and here the more complex flowchart of the question & answer program may show as a drawback. For instance, if we were to modify, or include, a new possibility in the interaction state diagram, it might give cause to a major redesign, or more likely, result in a "patch", eventually with a messy flowchart as consequence.

Segmentation and locality properties are similar in that they are reflected in the form of the flow chart. The many calls to I/O-routines mixed with computations indicate that these two properties are likely to be weak points when the pros and cons are to be counted.

Command dialogue

The portability of a command dialogue program will to some degree depend on the programming language used. The special purpose I/O-routines that are used to read and decode commands will include character manipulation. If this can not be done in the programming language, these parts will be computer dependent.

Maintainability and expandability have prospects for good values. The flowchart exhibits such simplicity and overall uniformity that modifications and expansions are simple and safe. For instance, a change in one of the commands will influence only the decoding portion of that part of the flowchart, while the inclusion of a new command or subcommand just will add a new path.

Likewise, the flowchart gives direct information on how to segment the program, or how to order the modules so as to achieve good locality.

3.9 Interactive User Categories

The users of an interactive program will differ in the relative importance they attach to the facilities offered. They also differ in the frequency with which they utilize these facilities. When designing an interactive program, it is of cource important to realize what the intended users will expect from it. The following is an attempt to summarize a few possible user categories together with their typical needs:

- the batch user
- the experienced user
- the casual user
- the beginner
- the assistant

The batch user can (and must) select in advance a sequence of actions that the program is going to follow, with a specified set of inputs.

It may sound strange that an interactive program might be used in batch mode. It is, however, not at all unnatural. In many cases, a set of similar problems is to be solved. The first two or three may with advantage be solved interactively. After that, the proper way of solving the remaining problems may be known and interaction is no longer valuable. Indeed, it may cause additional costs, as it requires constant human intervention and a more expensive way of running the computer. Therefore an easy and efficient way of running the interactive program in batch mode would be useful.

The experienced user is the one with the most exacting demands on the interaction. He has good knowledge and intuitive feeling for the methods he is using and knows and uses the facilities the interaction offers. He might be trying to solve a new and complicated problem exercising his prior knowledge, skill, intuition and common sense combined with the data handling power of the computer. In this situation he wants a maximum of freedom in the choice of solution steps. It is of great importance to him to be able to view the results from the computer as they become available and to be able to communicate his desires promptly. He is likely to be able to spot an erroneous or uninteresting result at an early stage and should have the possibility to abandon such an unpromising road.

The casual user could typically be a student solving a laboratory exercise. He would then have to solve a well-defined problem, which is known to be solvable by means of the program in question. Being a casual user, he would not be particularly interested in anything but the facilities necessary for his task. He would consider it an extra burden to be forced to learn a list of commands and command syntax, although this could be considered an advantage from didactic reasons. Rather, a dialogue

offering guidance would be preferred, decreasing the risk of serious mistakes and lessening the burden of the supervisor.

The beginner is initially in the same situation as the casual user, simplicity is important to be able to get started. There is a distinction though; the beginner has a desire to become an advanced user some day. He is interested to learn what facilities are available and to master them. He would want facilities for help and instruction and if possible, a means of gradually growing accustomed to the details of the program.

The assistant is someone that performs routine investigations, typically designed by the experienced user. The assistant is not required to know the fine details, neither of theory nor of the program. He is primarily engaged in providing the program with proper data and collecting the results. The means by which the experienced user instructs his assistant and whether or not primitives can be constructed is of great importance.

Naturally, the ideal type of interaction is quite different for these users, ranging from no interaction in the batch user case to the heavy demands of the experienced user. It is most important to realize however that the same program may have to meet these varying requirements. A few examples on this situation will be given.

First of all, the ideal situation for the beginner has already been described as a gradual change from interaction with much guidance to the full freedom of the experienced user.

Secondly, let us regard the casual user in the form of a student doing a laboratory exercise. Although he is using an interaction scheme with good guidance, he is likely to get stuck sooner or later. He will then call the help of his supervisor, presumably a more or less experienced one. In the correction of the student's mistake, the supervisor would prefer a more direct form of interaction.

Thirdly and finally, the experienced user may take many shapes. He may turn into a batch user if he finds that the interaction of a part of his job will be entirely predictable like in a Monte-Carlo simulation situation. Or he may be preparing primitives for routine investigations to be done by himself or by his assistant. Or, after a few months of disuse, he may be regarded a fast-learning beginner and will appreciate some of the informative functions created for the beginner.

Summarizing, it may well happen that the desired type of interaction is very varying and in the design of a program, the satisfaction of these demands will pose some problems.

3.10 The Question of Initiative and What do We Choose

In the choice between a question & answer dialogue and a command dialogue interaction, there is also a consideration of a philosophical nature. It could be formulated as the "Question of initiative" or "Who is who's slave?".

The cook-book engineer

The cook-book engineer functions with the help of tables, handbooks, and ready-made design procedures, maybe in the form of programs. The routine and exclusive use of these aids is likely to cause a loss of the intuitive feeling for the soundness of results. Similarly, knowledge, if there ever were any, will soon be forgotten.

The cook-book engineer might be dangerous if set to seek a solution beyond the normal range of problems. He is apt to fail to realize when basic assumptions are violated, and may produce results which are only subtly wrong. A typical error is that quantities that normally can be neglected and therefore are not accounted for, may grow to be significant if results are based upon mere extrapolations. Examples, sometimes wellknown ones, can be found in many disciplines of engineering and in society in general.

The quality of life

The industrial revolution and the resulting general prosperity was the result not only of mechanization but also of the specialization of jobs. The ultimate result is the assembly line organization where workers perform the same few movements hundreds of times a day. The resulting low interest in the work has created some interest in the re-organization on the workshop floor to encourage a deeper envolvement.

There is a danger that today's work in designing interactive programs for analysis and design will invite the same development in our own area of interest. Let us try to prevent the control engineer of tomorrow from simply becoming an input device to a computer with some superprogram. The task in our program design is thus to promote the use of the computer as a tool, so that the control engineer still can be the master not only of the machine but also of theory.

Thus we find a possible conflict between the task of making advanced methods easily available and simple to use, and the interest of an active knowledge of the underlying theory. Questions of this type has been discussed and elaborated in e.g. [Rose75] and also in general literature, among many examples in [Asimov].

Our favorite user

Our favorite user will thus be the one with a deep understanding of what he is doing. That implies we are to avoid setting restrictions to his use of program facilities, and where defaults are used it should be apparent when and how such defaults should be reconsidered. The user should be encouraged to take the initiative, and hints from the program as to possible future actions should be avoided, simply because of the risk that other choices might be more awarding in some special situations.

Our favorite user is in other words the one that in the previous sections were called the experienced user. The proposed concentration on the experienced user should not be construed as a recommendation that the other cathegories of Section 3.9 should be neglected. Rather, the task is to construct a program with means to take care of all cathegories. This can be done as will be demonstrated in the next two chapters.

The choice

The conclusion of this chapter is then that an interactive program should use a command dialogue. The reasons are several. In the sections on the influence on program structure, the command dialogue solution got higher or equal marks in all respects. The same applies for the comparison of Sections 3.4 and 3.6, with one exception (d). The question & answer dialogue makes possible a closer contact between the user and the algorithm he is using. This is because the input of parameter values and the like can be delayed until the information is about to be used. In principle the same effect may be achieved for the command dialogue through an appropriate division of the algorithm into many minor commands (subcommands). This would however be impractical and contrary to other requirements. In the practical use of an interactive program, however, the interest is not as much on the actual algorithms and their detailed behaviour, so this advantage of the question & answer dialogue may be of minor importance.

The command dialogue suits the experienced user, but leaves e.g. the casual user or the beginner entirely on their own. The next chapter will among other things show how this deficiency may be eliminated by inclusion of the 'interaction structure' feature already mentioned.

4. THE COMMUNICATION MODULE INTRAC

This chapter has three closely related themes. The first one is the description of an actually implemented communication module, Intrac [Wies78].

Intrac is a subroutine package designed so that it can be easily combined with application modules to form an interactive program. Intrac itself contains no application dependent features, so it can be used in any application field. In Idpac it has been supplemented with tools for identification and data analysis. Intrac is the main topic of Section 3 where its internal structure is discussed. The required interface to Intrac in the application modules is discussed in Section 4, while Section 6 describes the major application independant commands available within Intrac.

The second theme is of a more general nature. Here we explore the effect of a communication module like Intrac. In Section 1 the command structure and the data base are described, indicating the general form and philosophy of the command interaction available through Intrac. In Section 2 the effect on the overall program structure is examined. Section 5 finally presents Intrac as a basis for an interactive problem solving language. The material in these sections strongly depends on Intrac in details, but many other forms of a communication module could be concieved, providing the same basic facilities and influencing the host program in the same manner.

The third theme is found in Section 7. Here the concept of a problem solving language is exploited. It is shown that the possibility to write procedures, here called macros, in this language provides a considerable freedom. This freedom can be applied both in the practical use of the language and in the recasting of the interaction to suit various situations. The ideas of this section are completely general in the sense that the facilities and possibilities mentioned are natural consequences of the command dialogue approach to interaction.

4.1 Command Structure and the Data Base

A command in Idpac has the generic form:

<command identifier><argument list>

The <argument>s in the <argument list> convey information on the problem or its solution:

<argument>::=<integer>/<real>/<identifier>/
<delimiter>/<variable>

A few examples will serve as illustrations of the command form adopted for Idpac. The command (written following the prompting character >)

```
>PLOT DATA
```

consists of the <command identifier> PLOT and a single argument that is an <identifier>. PLOT signifies the action desired, viz. to draw a diagram on the display output, and DATA is the name of a set of values we want to visualize. An alternative form of this command could be, cf. the appendix:

```
>PLOT (100) DATA -5. 10.
```

Here we actually have six arguments, an <integer> is enclosed within parentheses, i.e. two <delimiters> and the argument list is ended by two <real>s. The effect of this extra information is to specify that 100 values should be plotted along the horizontal axis, and that the vertical axis should be scaled with -5 and 10 as minimum and maximum respectively.

Another example from Idpac is:

```
>ML (SC) MODEL = DATA 2
```

The effect is to specify a maximum likelihood identification on the data set DATA, producing a model of order 2 stored in the data base under the name of MODEL. The <identifier> SC, delimited by parentheses, is a flag saying we want to enter a subcommand sequence further specifying the actions to be taken. The <delimiter> = has an important function as 'syntactic sugar'. It is used to divide the arguments into an input part and an output part. Thus it is an aid for the computer in decoding, and for the human in memorizing and understanding the command line.

In Idpac, the use of an <identifier> is often interpreted as the name of a set of data, to be acted upon or to receive the results. In the current implementation, the data base is located on mass memory, and the names refer to files. This is, however, not at all the only possible way to do it. An <identifier> could equally well be regarded as a pointer in a data area contained in the address space of the program itself. In a properly structured program, like the one in question, the decision whether to look in primary memory or mass memory lies in the data interface routines, and their abilities is under the implementor's control.

Important is, however, that the data structures referenced in the <argument list> are properly defined and standardized. In this way an inter-command and inter-program compatibility is achieved that is valuable. This is the topic of Chapter 5.

4.2 Overall Program Structure

An interactive program built around Intrac will in principle have the structure shown in Figure 4.1, which illustrates the logical relationships between program modules, Intrac, the data base, and the user at the terminal.

There is a program module 'Main' which calls a number of subroutines, AR1, AR2, ... ARN, called action routines, and Intrac. When Intrac is called it will (normally) read a command line from the user's terminal and analyse it. The index of the received command in a command table sent to Intrac from the main module is found, and the rest of the command line is processed and stored in memory. If the command received is an application command, Intrac returns to the main module which in its turn uses the command number to select the proper action routine to call. Thus each application command generally corresponds to a specific action routine.

Intrac has the possibility to read the commands off mass memory, this is the case when a macro is to be executed. This will be described later.

The parallellism of the action routines is a noteworthy feature of the program. This reflects the situation of a state diagram with a single state, as is shown in Figure 3.4. The dashed lines in Figure 4.1 illustrate the flow of information when the arguments in the command line are analysed by the appropriate action routine. Note here, that when the command line has been analysed and the computation started, Intrac is no longer needed. This property may be used to segment a program so that computation code and Intrac share the same area in memory. This is important on small computers.

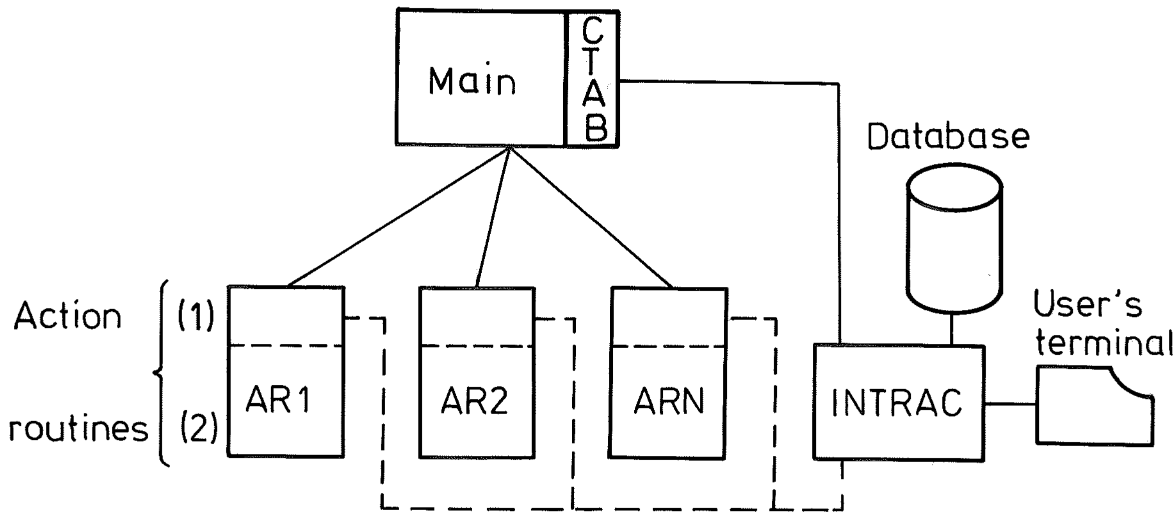


Figure 4.1 The principal structure of a program built around Intrac.
(1) indicates the command analyzing part.
(2) indicates the computing part.

It is possible for the selected action routine to call Intrac too, in order to receive commands to further specify the required action. Such subcommands have to be fully implemented within the corresponding action routine, i.e. it has to include the same things as the main module. This facility is discussed together with the action routine structure in Section 4.4. This is no great difficulty, however, since the main module is quite simple in structure, cf. Figure 4.2. The definition of the command table, i.e. a table of the legal application command names, is usually done using a DATA-statement, and the following initialization refers to data in Intrac and the application routines. The following call to Intrac causes a command line to be read. After processing it, Intrac returns with the index of the received <command identifier> in the command table. In case of a formal error, an error message routine is called. The index, ICMND in the figure, is used in a CASE-type statement (FORTRAN computed GOTO), to call the

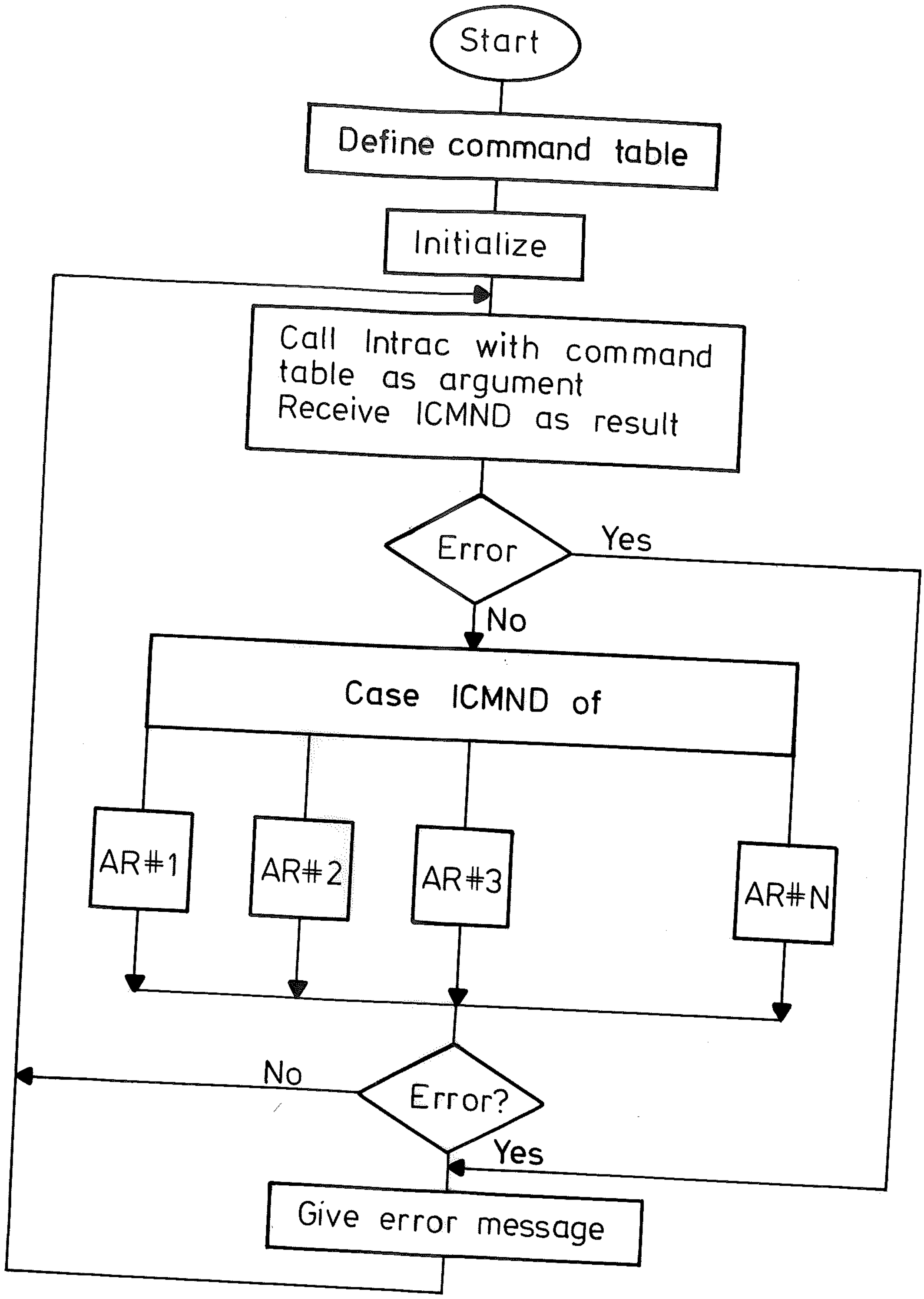


Figure 4.2 The main module structure.

appropriate action routine. After the return therefrom, a test is made to see if an error message is due, and the program is ready for the next command line.

Note finally, that not all commands entered on the terminal correspond to an action routine. Some commands of a general nature are processed entirely within Intrac itself. Others are interpreted as calls to macro commands. Both these cases are treated in the following section.

4.3 Sketch of the Structure of Intrac

Figure 4.3 gives a brief sketch of the internal structure of Intrac. A call to Intrac causes a command line to be read from the current input device, normally the user's terminal. In COMLIN, the command line is divided into its constituent parts, i.e. the arguments (if any) are recognized as <identifier>s, <integer>s, <delimiter>s etc., and their values are stored as a vector together with their respective type. The use of this argument vector is described in the following section.

Only the first item in the command line is used by Intrac, namely the <command identifier>. Intrac implements a set of application independent general purpose commands. Intrac now interrogates the table containing their names. If a match is found, the routine RESEX is called upon to carry out the desired action, otherwise the <command identifier> is compared to the entries in the table of application command names sent to Intrac by the calling routine. An example is shown at the top of Figure 4.2. If a match is found here, we know that an application command was requested and the caller will receive an index specifying which action routine to call. Before the return, however, the routines SUBST and RECLIN are invoked.

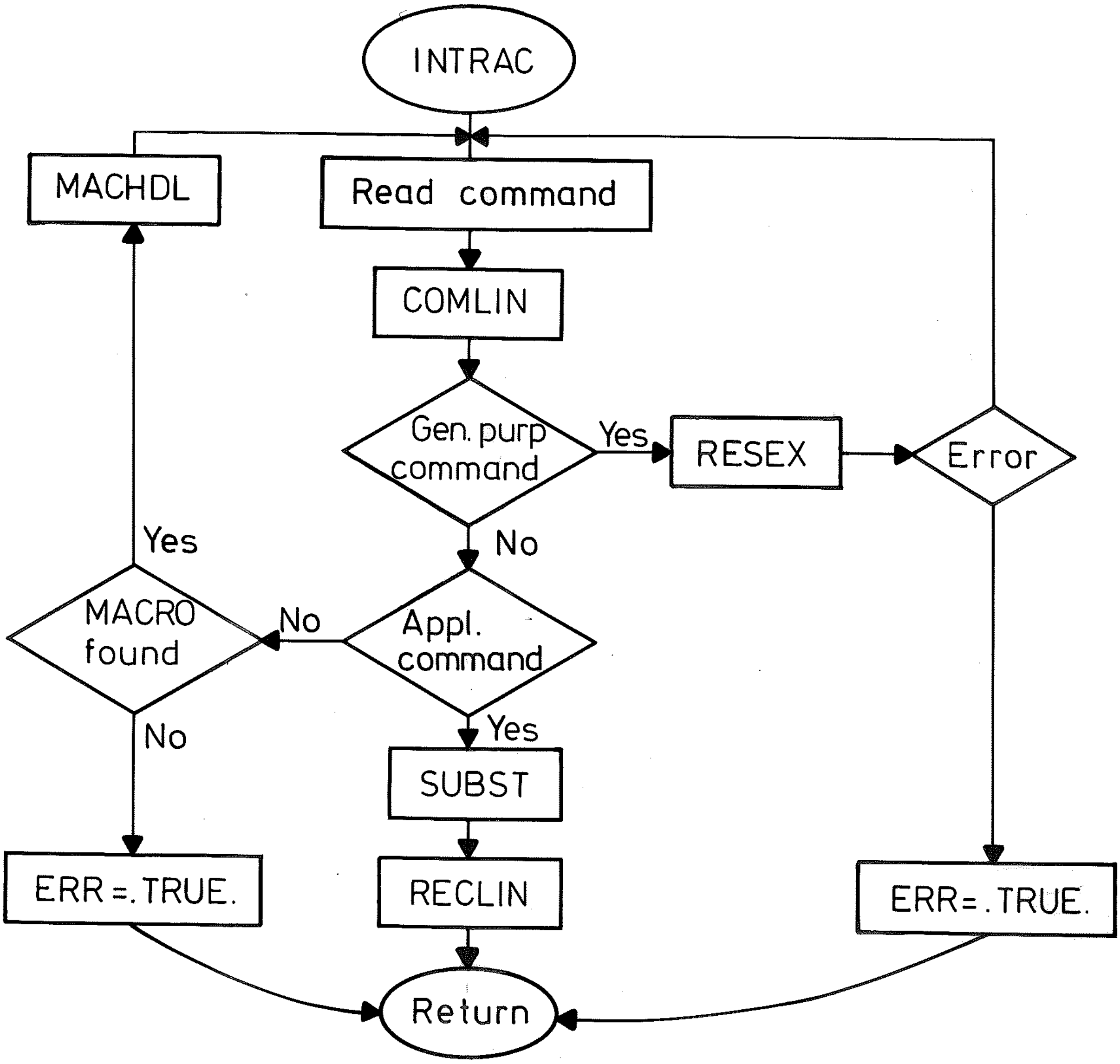


Figure 4.3 The internal structure of Intrac.

RESEX includes code to execute the general purpose commands LET, FOR etc. detailed in Section 4.5.

SUBST will take every occurrance of an <identifier> among the arguments and check if it is the name of a <variable> known to Intrac. If this is the case, the <identifier> is substituted by the value of that variable. Intrac (and SUBST) can do this because it maintains an internal table of variable names and values. Variables are discussed at greater length in Section 4.5. Due to the substitution in SUBST, the actual command line seen by the action routines is not the one initially read by Intrac. To allow the actually processed command line to be output to a log, RECLIN will reconstruct a text string with the same effect as the one seen by the action routine. Thus if V is the name of a variable with the value 2.Ø, the command

    CMND V

will in the log look like:

    CMND 2.Ø

Finally, if the <command identifier> is not found in the command table, the data base is searched for a macro, Section 4.5, with that name. If one is not found, the <command identifier> is illegal, i.e. it does not correspond to an action known to Intrac at that time. If a macro is found, the macro handler MACHDL is called. Apart from changing the input device for the command read operation, it sets up the correspondence between actual and formal arguments in the macro call.

The change in input device causes the following command lines to be read from the data base. Simple as it is, this idea allows complicated actions to be stored in the data base and invoked in a form entirely like a single although very powerful application command, hence the name 'macro'. Apart from the actions in RESEX, the operations within Intrac are fairly straightforward. Note that as long as no errors are found, general purpose commands, i.e. those executed by RESEX, are treated entirely within Intrac regardless of subcommand levels etc.

4.4 Action Routine Structure, Decoding Primitives

Error detection and error recovery is an important aspect of interactive programs. THis is clearly reflected in the flow diagram of an action routine as is shown in Figure 4.4.

Figure 4.4 shows the principal logic flow of an action routine. After an initialization, the argument list is decoded using primitives from Intrac. The details are described below. If an error is detected an error indicator is set and the routine terminates. Otherwise a flag is tested to see whether the command actually should be executed or not. If not, LPCOM is called to allow a command log to be produced. This mode serves to decode command lines with a check for possible errors during the generation of a macro, without a command being executed. If it is to be executed, which is the normal case, and also a possibility during macro generation, we start reading input data (if any). After having started opening files, it is necessary to keep track of them so that they are closed, also in the event of errors detected. When inputs have been read without errors, the main algorithm is applied. When it has been completed without errors, still it is necessary to check that it is possible to output <u>all</u> results before actually doing so. This is to ensure that the data base is kept consistent, i.e. to avoid modifying one part of a dataset while a second part is left unaltered because of the late detection of an error. Finally, the command line is logged only if no errors were detected and before any output is produced.

The decoding of the argument list is an important aspect of the action routine. The arguments in the command line were extracted in Intrac by the routine COMLIN and placed into a vector (or rather several in parallel in the sense of FORTRAN, cf Section 6.5c), containing the value of the i:th argument and its type (<delimiter>, <identifier>, <integer>, or <real>).

This 'argument vector' is accessible from the action routine. To simplify the programming of action routines, Intrac contains a set of logical functions called command decoding primitives. These will aid considerably in decoding and checking the argument list.

The primitives have the form of logical functions that look at a specified position in the argument list and return the value true if the item is of the desired type. As a side effect the 'function' returns the actual value of the argument through the function parameters. This idea is best shown via an example. It is taken from Idpac and contains the essential code from the argument decoding in the command FILT, see Figure 4.5.

Example

The command syntax for FILT is

    FILT SYST = TYPE NO T OML [OMH]

The left hand side has a single argument; the name of a dynamic system to be generated. That system should be a Butterworth filter specified by the right hand side; low-pass, band-pass, or high-pass determined by the flag TYPE. The order of the filter is specified by the integer NO. The sample time is given by T and the cut-off frequency by OML. In the case of a band-pass filter, the high cut-off frequency is given by a fifth right-hand side argument, OMH.

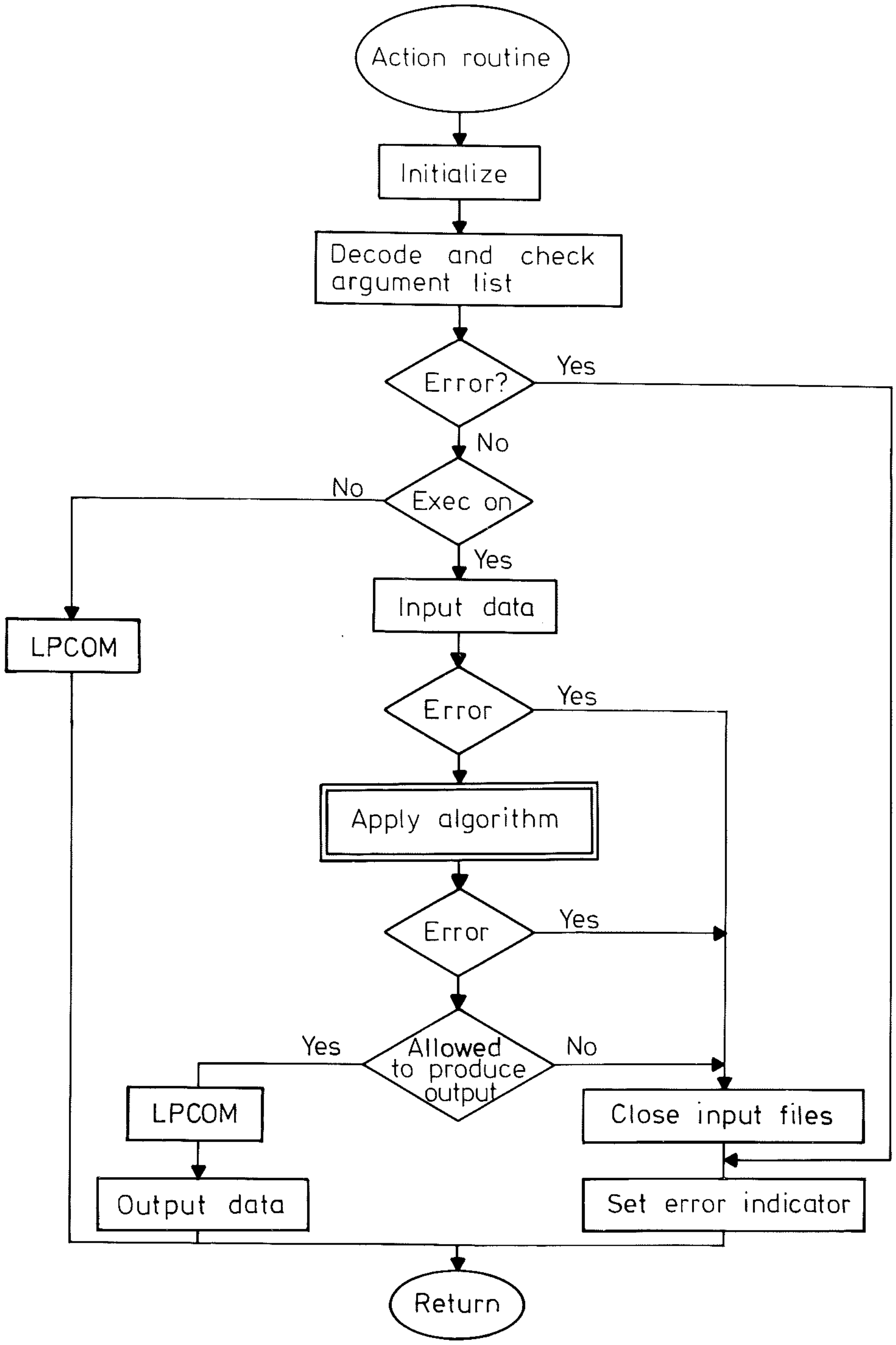


Figure 4.4 The structure of an action routine. Only the algorithm and the I/O data is application dependent.

The code example in Figure 4.5 starts with the definition of three text constants 'LP' etc. and two integers NRL and NRR belonging to a common block. The two integers are returned from COMLIN and indicates the number of arguments to the left of the equal sign (including the <command identifier>) and the number of arguments to the right respectively. The two arithmetic ifs thus give an initial test on the argument list.

The detailed examination starts in the second (ICNT=2) position in the command line. Here an <identifier> is expected. The call to the function LHNAME will return the value <u>true</u> if this is indeed the case, otherwise <u>false</u> is returned, followed by a GOTO with the result that the error message 'BAD FILE NAME' is output. As a side effect of a successful call, ICNT is incremented to point at the next argument and SNAM receives the value of the <identifier>.

The first arguments on the right hand side are then decoded in a similar fashion. The type of filter is obtained through a call to LHOLLS, which compares the actual argument with a list of three alternatives. If a match is found, The integer IFILT tells us which alternative, otherwise an error 'BAD FILTER TYPE' is produced. The filter order is found with the help of LINT while the sample interval and the first cut-off frequency is found using LNUMB. LNUMB accepts a number i.e. either an <integer> or a <real> argument. Finally, the correct position of the terminator in the argument list must be checked. For a case other than the band-pass filter, we should reach the terminator after the fourth argument, otherwise a fifth argument should be allowed.

This fairly detailed example shows how the command decoding is actually implemented in programs using Intrac. The required type of arguments is represented by the choice of decoding primitive, while the structure of the argument list

```
      SUBROUTINE FILT
C
      DIMENSION TYPES (3)
      COMMON /COMINF/ NRL,NRR, ...
      DATA TYPES /'LP','HP','BP'/
C
      IF (NRL-2) 500,100,510
100   IF (NRR-4) 550,105,105
105   ICNT=2
C     OUTPUT FILE NAME
      IF (.NOT. LHNAME(ICNT,SNAM)) GO TO 520
C     FILTER TYPE
      IF (.NOT. LHOLLS(ICNT,TYPES,3,IFILT)) GO TO 580
C     FILTER ORDER
      IF (.NOT. LINT(ICNT,NO)) GO TO 530
      IF (NO .GT. MAXO) GO TO 540
C     SAMPLE INTERVAL
      IF (.NOT. LNUMB(ICNT,DELTAT)) GO TO 590
C     CUT-OFF FREQUENCIES
      IF (.NOT. LNUMB(ICNT,OML)) GO TO 590
      OMH=OML
      IF (IFILT .EQ. 3) GO TO 180
      GO TO 200
C     GET OMH
180   IF (LTERM(ICNT)) GO TO 550
      IF (.NOT. LNUMB(ICNT,OMH)) GO TO 590
200   IF (.NOT. LTERM(ICNT)) GO TO 560
C     START COMPUTING
C
C     ERROR CONDITIONS
500   TOO FEW LEFT ARGUMENTS
510   TOO MANY LEFT ARGUMENTS
520   BAD FILE NAME
530   BAD INTEGER
540   FILTER ORDER TOO HIGH
550   TOO FEW RIGHT ARGUMENTS
560   TOO MANY RIGHT ARGUMENTS
570   A FREQUENCY IS OUT OF RANGE
580   BAD FILTER TYPE
590   BAD NUMBER
```

Figure 4.5 The essentials of the command decoding in FILT. The error condition statements are indicated in a condensed form.

is represented by program code. The task could be solved more elegantly in a programming language other than FORTRAN. The ultimate solution is however a new level of interactive programming language, usable both in the implementation of application software (action routines) and directly in the daily use of the program, cf. Sections 5.5 and 6.7.

4.5 Facilities in the Intrac Language

The user of an interactive program based on Intrac interacts with the program via a terminal and expresses wishes concerning the solution of his problem in the form of commands or answers to questions. The commands can be divided into different categories. Some general purpose commands (or Intrac statements) are handled by Intrac itself, while others, application commands, are analyzed by Intrac and then passed on to the main program module which selects the appropriate action routine to handle them. The form of the commands was exemplified in Section 4.1. In some cases the action routines may need or offer further interaction in order to carry out the desired actions. This is then accomplished by means of subcommands, i.e. commands received through Intrac but with a different command table, depending on the specific action routine.

The Macro

Macro commands is another facility supported by Intrac. They are calls to previously defined command sequences on mass memory. Technically, when Intrac recognizes a reference to such a command sequence, it starts reading commands from a mass memory file, rather than from the user's terminal. A macro corresponds to subroutines or procedures in ordinary programming languages.

A macro consists of a sequence of Intrac-statements, macro calls and application commands. They are stored as a text file on mass storage. The first line in the macro should be a MACRO-statement which has the following form[1].

```
MACRO <macro identifier> [<formal argument>/<delimiter>/
                           <termination marker>]*
```

The statement declares the formal arguments of the macro. After the MACRO-statement follows a sequence of Intrac-statements, macro calls and application commands. The last line in the macro should contain an END-statement:

```
END
```

A macro is called by giving its name followed by actual arguments in the same way as a command. If the <termination marker> is not used then the number of actual arguments should be equal to the number of formal arguments in the MACRO-statement. The delimiters appearing among the formal arguments should be given at corresponding positions in the call.

The <termination marker> is used when a variable number of actual arguments is allowed. It indicates that the formal arguments and delimiters appearing following the symbol need not have corresponding actual arguments. If the <termination marker> is used several times in the macro, then it gives alternative places where the call can be terminated. The formal arguments which have no corresponding actual arguments will be 'unassigned'.

1) The notation [ ]* denotes that the enclosed item is optional or may be repeated several times.

Intrac as implementing a language

Actually, a program built around Intrac may be regarded as an interpreter for an interactive problem solving language with the same type of facilities as found in many other languages for interactive programming. An important difference is, however, that here we are aiming towards a specific problem area, by the inclusion of special problem oriented action routines / commands. Macros (subroutines / procedures) in this problem oriented language can be used to implement common subproblem solutions, give user guidance or implement question / answer dialogue. In this way many of the demands mentioned earlier in Section 3.1 can be met. In order to emphasize Intrac as the basis of a language, its data handling capabilities are discussed in the following subsection. The constituent statements of Intrac are listed in Section 4.6. A detailed account of Intrac is found in [Wies78].

Variables

In Section 4.1 we saw that a <variable> was a possible item in an <argument list>. A variable can be of three different types:

```
<variable>::=<formal argument>/<local variable>/
             <global variable>
```

where

```
<formal argument>::=<identifier>
<local variable>::=<identifier>
<global variable>::=<identifier>.[<identifier>]
```

The value of a variable can be either an integer number, a real number, an identifier or a delimiter. A variable can also be unassigned.

When processing the argument list of a command, Intrac will try to substitute a value from its internal tables for every occurrance of an <identifyer>. This was described in detail for the routine SUBST in Section 4.3. The substitution rule does not always apply to Intrac-statements. The items which can be substituted are underlined in the syntax for the Intrac-statements. With these exceptions, the substitution enforces what in common programming languages is called 'call by value'. Although quite appropriate in most cases, this compulsory substitution is too restrictive in some cases. If Intrac ever is to be redesigned, the substitution rule should be made conditional. It would then be possible to return values from a command through its arguments. The potential is that such results could be used in conditional GOTO statements, making the future actions of a macro dependant on the results. This facility is currently available using global variables.

The arguments listed in the definition of the macro are called formal arguments. When the macro is called, a corresponding list of actual argument values should be specified. The substitution rule is then applied so that every occurence of a formal argument in the macro is replaced by its value before the command arguments are passed on to the proper routine.

A local variable has the same form as a formal argument and it is in fact treated in very much the same fashion. It is local to the macro level and is defined when it is first given a value in a READ, FOR, LET, or DEFAULT statement.

A <global variable> is distinguished by a dot following the identifier. It is always accessible and may be used to pass information between macros. An important use is to define a set of problem dependent parameters stored as <global variables> that can be referenced in several different but related application commands or macros. The value of <global

variable>s may also be used and returned directly by application command routines. In fact, <global variable>s are the only means by which results may be returned from application commands within the framework of Intrac. Other possible ways, files, data areas, etc. are not administered by Intrac. It should also be mentioned, that the implementor of an interactive program package has the possibility to initialize the table of <global variable>s so that there always will be a set of "reserved" <global variables> for special use.

Under certain circumstances, a global or local variable, or a formal argument may have been defined without having been assigned a value. The type 'unassigned' may be transferred in a LET-command. The action on 'unassigned' values by IF...GOTO commands is defined, and most importantly, the DEFAULT command is specifically designed to handle them. If an unassigned variable appears as an argument to an application command it will be totally invisible to the corresponding routine.

4.6 Intrac statements

Intrac implements a number of statements of an application independant nature. They provide many of the functions found in any general purpose programming language. They therefore further emphasizes the idea of Intrac as a basis for application oriented problem solving languages.

a) Generation of macros

There are some different ways to generate a macro. Since a macro is implemented as a text file it is possible to generate and modify a macro using a text editor. A macro can also be generated by entering the MACRO-statement from the terminal. This statement was defined in the previous section. In generation mode all correct commands entered from the terminal are stored on a file. This continues until generation mode is left by the END-statement. Whether the commands in the macro should be executed during generation or not is determined by the switch EXEC. If EXEC is OFF then the commands are only checked for formal errors and if correct stored on the file. If EXEC is ON the commands will also be executed.

The FORMAL-statement can be used to extend the list of formal arguments anywhere in the macro. It is placed after the MACRO-statement automatically when the generation is finished.

b) Assigment of variables

Formal arguments are allocated and possibly assigned when a macro is entered. Their values can be changed with the LET-, DEFAULT-, FOR-, and READ-statements. Among the forms allowed is the usual arithmetic statement, and the main form is:

LET {<variable>=}* {<u><number></u>[{+/-/*//}<u><number></u>]

The DEFAULT-statement is a conditional assignment statement. Its form is:

DEFAULT {<variable>=}* <argument>

The assignment is performed only if either
- the named variable is 'unassigned'
- the named variable does not exist.

In the last case a new variable is allocated.

c) Branching

To make macros flexible it is necessary to have a way to change the sequence of commands executed. This may be acieved through branching statements and labels. The labels used in branch statements are declared 'on site' using the LABEL-statement:

LABEL <label identifier>

<label identifier>::=<identifier>/<integer>

The unconditional GOTO-statement is:

GOTO <label identifier>

Since the argument in the GOTO-statement could be a variable whose value is a label identifier it is possible to use the statement as the assigned GOTO of FORTRAN.

The conditional GOTO statement has the form:

IF <argument> {EQ/NE/GE/LE/GT/LT} <argument>
  GOTO <label identifier>

The effect of this statement is the same as for the GOTO-statement if the relation is true. If it is false the next command in sequence is executed.

d) Looping (FOR, NEXT)

It is possible to introduce loops among the commands in a macro. This is done with the FOR- and NEXT-statements. The FOR-statement begins the loop and has the following form:

FOR <variable> = <u><number></u> TO <u><number></u> [STEP <u><number></u>]

The NEXT-statement ends the loop and has the form:

NEXT <variable>

e) Output and input

The macro facility can be used to implement question and answer interactive programs. Questions are written on the terminal with the WRITE-statement and the answers are read using the READ-statement. The WRITE-statement is used to write variables and text strings. Its form is

WRITE [([DIS/TP/LP] [FF/LF])] [<variable>/<string>]*

The READ-statement reads values from the terminal and assigns variables. Its form is:

READ { {<variable> {INT/REAL/NUM/NAME/DELIM/YESNO}} /
<termination marker>}*

After each variable a type specification for the expected value is given:

```
INT     - integer number
REAL    - real number
NUM     - integer or real number
NAME    - identifier
DELIM   - delimiter
YESNO   - identifier YES or NO
```

When the READ-statement is executed a prompting # is written on the terminal. The <termination marker> has the same function as in the MACRO-statement. It gives alternative places where the answer could be cut off. The variables that are not given any value become 'unassigned'.

There are two means of escape from the READ statement, resulting in the suspending of the macro.

- If the answer is just a > the READ-statement will have no effect and the macro is suspended. If the macro is resumed by the statement RESUME the READ-statement will be re-executed.
- If an acceptable answer is given followed by a > the variables will be properly assigned and the macro suspended. If the macro is resumed with RESUME, the command following the READ will be executed.

f) Suspending a macro

There are cases when the freedom to have formal arguments in a macro is not enough. At generation time it may for example not be known which command is appropriate at some point in a macro. It is then possible to switch to command input from the terminal (i.e. suspend the macro). When the command input from the terminal is finished the macro is resumed. This facility is handled by the statements SUSPEND and RESUME.

A macro is automatically suspended in some cases.

- When an error is detected during the execution of a macro then an error message is printed and the macro is suspended. The user can then e.g. enter a correct form of the erroneous command and then RESUME the macro.
- When the READ-command has been executed in a macro, the user has to input the requested values from the terminal, or he can enter a special escape character (>) which causes the macro to be suspended.

4.7 How to Use the Macro Facility

The macro facility is based on a simple and basic idea, viz. that a character string is interpreted as a name of a body of text that is to replace that string. This is the macro concept found in many assembly languages for computer programming. In Intrac the effect is achieved simply by altering the input device so that commands are read from a mass storage file. An assembly language macro offers the possibility of arguments, which are treated as text strings that replace the occurrance of the argument in the macro body. Macros in Intrac also allows arguments, although they are replaced with their values rather than their text representation. This was a more natural way to go, since it is values that are ultimately passed on to the action routines. Also, values are easier to handle, since they have constant "length", i.e. they are stored in a known number of locations in memory, as opposed to text strings.

Having come this far, it is natural to realize that the inclusion of structural statements like branching and looping, simply boils down to a search for specific positions within the text file. Likewise, being able to handle values for actual arguments, it is natural to allow variables local to a macro, and I/O statements that transfer variable values. In short, the facilities included in Intrac and described above are natural extensions to the basic command philosophy. The rest of this last section will show that they are indeed also useful.

In the context of Chapter 3, the Intrac language is primarily suited for the needs of the experienced user, giving access with few restrictions to all available commands, in any order. As indicated there, this complete freedom may not always be desirable, so other forms of interaction should be provided. This can be done by means of the macro facility, and we will now demonstrate the main ideas in how to obtain the desired result.

Commonly used command sequences

The experienced user will often find that a command sequence is frequently executed with only minor changes. It is then convenient to introduce that special command sequence as a macro. This procedure can also be thought of as a mechanism for generation of new commands, suitable for a specific problem. This case is illustrated in Chapter 8.

This type of macros may serve as a short hand facility for the experienced user, and as simple-to-use primitives for his assistant. In the examples, the macros were generated in the mode EXEC OFF. The mode EXEC ON is suitable when, during the solving of a problem, it is apparent that the following actions will be used more than once. By starting the definition of a macro during the first time through, the command sequence is then immediately available for repeated use.

Simplified command forms

Figure 4.6 shows a method of implementing commands with two possible call formats. One form allows a single line call with arguments, while the other form consists of only the command name. The necessary arguments are then asked for, one by one. Finally, the proper action routine is called. This is the mixed dialogue mentioned in Section 3.2. The reason to use the mixed form is that a simplified interaction may be better suited for the infrequent or casual user. This example demonstrates a possible implementation of the command EIGEN available in Synpac and Modpac. The actual computations are supposed to be implemented in the action routine called by the command QREIG (eigenvalues by the Q-R method).

a) Note the use of the <termination marker> ; and the use of the 'unassigned' global variable UNASS..
b) Note the use of a <delimiter> in the list of formal arguments. The rules state that the same delimiter must appear in the same position among the actual arguments.

The command syntax for EIGEN as implemented by Figure 4.6 thus looks like:

EIGEN [<matrix of eigenvalues> <matrix of eigenvectors> =
<matrix identifier>]

```
MACRO EIGEN ; EVAL EVEC = A
IF EVAL NE UNASS. GOTO XCT
WRITE 'Name of eigenvalues?'
READ EVAL NAME
WRITE 'Name of eigenvectors?'
READ EVEC NAME
WRITE 'Name of matrix?'
READ A NAME
LABEL XCT
QREIG EVAL EVEC A
END
```

Figure 4.6 A macro implementing a command with mixed mode interaction. If no arguments are given, they are asked for one by one.

Question & answer interaction

A question & answer dialogue, giving good guidance for the infrequent or one-time user, may be realized in the form of a macro. A simple example using commands from Idpac is shown in Figure 4.7.

The READ and WRITE general purpose commands of Intrac are used to communicate with the user, presumably in this case a student of stochastic processes. Instead of just 'playing around' with some of the commands in Idpac, requiring some familiarity with specific details, he is taken in an orderly fashion by a macro through a sequence of commands showing the effect of a class of dynamic systems on a white noise input. Some points are worth noting:

a) If an error is detected, the macro will be suspended, i.e. the program goes into command mode. Any Idpac command is then legal. The inexperienced user is advised in the description to use GOTO RESTART, which will allow a complete description of filter parameters.
b) The use of the <termination marker> ; in the reading of cut-off frequencies allows input of only one real value.

```
MACRO NOISEDEMO
INSI WNOISE 200
NORM
X
LABEL DESCR
WRITE 'The effect of filtering white noise through'
WRITE 'Butterworth' filters will be demonstrated. You can'
WRITE 'choose filter type, order, and cut-off frequency.'
WRITE 'In the advent of errors, type GOTO DESCR to receive'
WRITE 'this description again, or type GOTO RESTART to'
WRITE 'start from the following.'
LABEL RESTART
WRITE 'Choose filter order and type (LP, BP, HP).'
READ N INT TYPE NAME
WRITE 'Now enter cut-off frequency. Enter two frequencies'
WRITE '(low and high) if you chose BP.'
READ CF REAL; CF2 REAL
FILT FILTR < TYPE N 1. CF CF2
DSIM COLNOISE < FILTR WNOISE
WRITE 'Hit the return key to see 50 samples of gaussian'
WRITE 'noise coloured by your choice of filter.'
READ ; I INT
PLOT 50 COLNOISE  "Plot of coloured noise
WRITE 'Hit return key to see Bode plot of theoretic and'
WRITE 'computed power spectrum.'
READ ; I INT
KILL
ASPEC NSP < COLNOISE 50
SPTRF (POW) FSP < FILTR B/A
BODE FSP NSP
WRITE 'Do you want another run?'
READ ANS YESNO
IF ANS.EQ.YES GOTO RESTART
END
```

Figure 4.7 A simple question & answer demonstration of coloured noise, implemented via a macro containing informatory text and questions.

The local variable CF2 will then be 'unassigned', and its appearance in the command FILT will be invisible to the action routine.

c) The dummy read statements READ ; I INT, where the ; allows the user to respond with an empty line, serves to include a pause so that the display is not erased until the user is ready.

Macros giving help and information

For the user with ambition to learn the possibilities of an interactive package in order to some day be an experienced user, facilities other than those above are needed. Also, the experienced user may need occasional short advice, e.g. on a seldomly used facility. For such purposes, a help facility is often made available. Here we will show that the macro facility well serves to implement such a function.

The macros given here (Figures 4.8 - 4.11) will be used as examples. HELP is intended solely for the use of the novice. It implements a form of programmed instruction where the student may choose when to change to a more advanced level of training. HELPSYN and HELPINF serve to write some informatory text, chosen through the argument. They are called by HELP, but used separately, they will prove useful also to the advanced user. HELPSYN will give the syntax of the command in question, while HELPINF will give information on the nature and use of the different command arguments. HELPEX, shown here giving an alternative to the example in Figure 4.6, will ask questions to execute a command.

The operation of the macro HELP is as follows: The beginner wanting to get to know the interactive program on his first session at the terminal types HELP as response to the promting character. The presentation and the information on the different modes of the help offered is then output. The mode will initially be Ø. The beginner should keep this value the first time and will then get the menu, i.e. a list of all available commands, shown. He then indicates his interest for one of the application commands, and then, being in mode Ø, receives information on the chosen command. In this way, a certain familiarity with the program is gained.

After a while, the user will feel ready to execute the commands. Specifying mode 1 in the section 'MODES' will cause HELP to execute the command chosen by the user in a question & answer type mode. Finally, the user will try his wings by writing the complete command with arguments. Mode 2 will still give some help in that the proper command syntax is displayed.

In this way, the beginner will recieve support according to his current state of training. Finally the user won't need the detailed help the HELP macro gives. The subfunctions HELPSYN and HELPINF will however still be useful also for the experienced user, and can of course be used separately.

Some parts of the macro HELP are worth further comments:

a) The DEFAULT statement in the beginning of the macro gives the possibility of initialization the first time the macro is called.
b) The repeated use of WRITE statements to output large amounts of text is somewhat clumsy. It is done here so that the text is possible to read in its context. In most implementations, there will probably be an application command available to output text files to a terminal. Here we use the command LIST from Idpac to output text describing the menu, i.e. a list of possible operations in a hypothetic application.
c) LIST has a possibility to output portions of a text file, utilized in HELPSYN and HELPINF. In fact, the mechanism used is the same as the one used to recognize sections within a system file, cf Figure 5.12.
d) Note the use of global variables to allow the mode and state of the macro to be saved so that at the next call, the desired options are still in effect.

```
MACRO HELP
"
" Demonstration HELP function.
"
DEFAULT HELP.STATE=Ø
IF HELP.STATE EQ  1 GOTO ACTIVE
IF HELP.STATE EQ -1 GOTO MENU1
"
" Initialize
LET HELP.STATE=1
LET HELP.MODE =Ø
"
LABEL PRESENT
WRITE 'PRESENTATION.'
WRITE 'This is a demonstration of some possible help'
WRITE 'function facilities.'
WRITE 'The help function can work in different modes,'
WRITE 'selectable in the MODES section.'
WRITE 'The help function utilizes functions that also can'
WRITE 'be called upon directly. They are:'
WRITE ' '
WRITE 'HELPSYN CMND  - Displays the command syntax for CMND'
WRITE 'HELPINF CMND  - Displays information on command CMND'
WRITE 'HELPEX  CMND  - Ask questions to help execute CMND'
WRITE ' '
"
LABEL MODES
WRITE 'MODES.'
WRITE 'You may now choose the mode of this help function.'
WRITE ' '
WRITE 'Ø - Obtain information only'
WRITE '1 - Obtain help to execute'
WRITE '2 - Obtain command syntax, execute by yourself'
WRITE '3 - Execute by yourself with no help'
WRITE ' '
LABEL CHOOSE
LET TMP=HELP.MODE
WRITE 'Choose mode by typing the appropriate integer.'
WRITE 'The current value is (' TMP '). You may accept'
WRITE 'this with an empty line.'
READ ; TMP INT
IF TMP LE Ø GOTO WRONG
IF TMP GT 3 GOTO WRONG
LET HELP.MODE=TMP
GOTO MENU
LABEL WRONG
WRITE 'Your answer must be in the range Ø-3'
GOTO CHOOSE
```

Figure 4.8a First part of a macro realizing a HELP function.

```
LABEL MENU1
LET HELP.STATE=1
LABEL MENU
LIST (T) MENU
WRITE ' '
WRITE 'MODES   - Change help mode'
WRITE 'PRESENT - Obtain the presentation of help'
WRITE 'EXIT    - Exit from the help function'
WRITE ' '
WRITE 'What is your interest?'
READ ANSWER NAME
IF ANSWER EQ MODES   GOTO MODES
IF ANSWER EQ PRESENT GOTO PRESENT
IF ANSWER EQ EXIT    GOTO EXIT
" Must be a request for an application command
" Act according to mode
IF HELP.MODE EQ Ø GOTO INFO
IF HELP.MODE EQ 1 GOTO XCT
IF HELP.MODE EQ 3 GOTO LEAVE
" Must be mode 2
HELPSYN ANSWER
LABEL LEAVE
WRITE 'Now you are in command mode.'
WRITE 'Return to HELP by typing RESUME.'
SUSPEND
GOTO MENU
"
LABEL INFO
HELPINF ANSWER
GOTO MENU
"
LABEL XCT
HELPEX ANSWER
GOTO MENU
"
LABEL ACTIVE
WRITE 'HELP is already active!'
WRITE 'Use RESUME to obtain more help.'
GOTO END
LABEL EXIT
" HELP not active any more.
LET HELP.STATE=-1
LABEL END
END
```

Figure 4.8b The second part of the HELP macro.

```
MACRO HELPSYN CMND
LIST (T) SYNTAX(CMND)
END
```

Figure 4.9 This macro is intended to give information on the syntax of a given command. It assumes that the informatory text is stored on a file SYNTAX with sections referencable as in Figure 5.12.

```
MACRO HELPEX CMND
GOTO CMND
WRITE 'There is no command with name ' CMND
GOTO EXIT
" Here code for other commands could be inserted.
"
LABEL EIGEN
WRITE 'Name of eigenvalues?'
READ EVAL NAME
WRITE 'Name of eigenvectors?'
READ EVEC NAME
WRITE 'Name of matrix?'
READ A NAME
EIGEN EVAL EVEC = A
LABEL EXIT
END
```

Figure 4.1Ø This macro shows how help to execute commands could be offered. Only one example is shown. Note the action taken for an illegal argument.

```
MACRO HELPINF CMND
LIST (T) INFO(CMND)
END
```

Figure 4.11 A macro exactly similar to the one in Figure 4.9 used to output informatory text.

5. DATA TYPES AND DATA STRUCTURES

An interactive program package is likely to be required to deal with data of many different types. Therefore it is an important task for the designer of such a package to find generally applicable methods to handle different data types.

The fist data type considered is scalars. A nice point with scalar data is that they do not occupy much space in storage, therefore they can be kept in tables in main memory. Still, some method of addressing them must be devised.

Nonscalar values (vectors & matrices and similar things) may take considerable storage space. They are therefore best stored on mass memory devices such as disc either directly or indirectly. The designer must then decide how these data types are to be structured. The easy way is just to dump the internal data structure onto a file, and then to read it back, when needed. If, however, such files are to be exchanged between program packages, a predefined yet flexible file format must be constructed. If the program is to be portable, the operations to open and close files must also be given some thought.

Finally, dynamical systems present some difficulties.They can be represented in many different ways., e.g. state equeations or input - output relations. In particular linear systems can be represented by a quadruple of matrices, by a matrix of rational functions, by polynomial matrices or by tables of impulse responses or frequency responses. It is desirable to allow many such representations simultaneously, and to be able to distinguish between them.

This chapter aims at presenting these problems in more detail and to review two possible solutions. The first four sections deal with the problems mentioned above. Then a skeleton implementation in a Simula like language is shown and finally details on an implementation with mass-memory files in FORTRAN is discussed.

5.1 Scalars

Generally speaking, scalars in an interactive program package are very often used to specify a problem and details on the method of its solution. Examples of integers of this kind are orders, iteration counts and indices. Reals are e.g. bounds, weighting factors, scaling factors and the like. The list can easily be made longer and more specific.

Character strings are a type of scalars of a different nature. The typical use is either as flags or as names. A flag is used as an indicator that one of a number of possible options should be used, the choice being made through the use of a string rather than an integer selector constant because of obvious mnemotechnical advantages.

Names can be used to identify data stored in various forms. The importance of using names rather than other possible ways (e.g. record numbers, memory addresses, indices in tables etc.) is the possibility of using mnemotechnically natural names of the user's own choice.

An interactive program based on Intrac (Chapter 4) will handle much data of scalar form, viz. the command arguments. Then scalar values of the above types are transferred from the command line. Also Intrac substitutes scalar values from its internal tables when they are referenced through their variable names.

On the whole, however, scalar values do not present great problems; there are no problems with structure and they occupy little storage.

5.2 Arrays

One or two dimensional arrays represent the simplest example of structured data. Here a set of elements of identical type (e.g. real numbers) are forming a pattern of rows and columns. This structure is characterized by one or two integers, viz. number of columns and maybe number of rows. Strictly speaking, this structure corresponds to vectors and matrices, but as we will see it is possible and useful to treat a number of other quantities in the same way.

Vectors and Matrices

Vectors and matrices are very basic data structures, essential in many methods applicable in automatic control. The most obvious use is in the description of linear multivariable systems on state space form. As such they are included in the discussion on system representations later on.

It may be interesting to handle matrices (vectors may be regarded as a special case) independently. This poses no problem since matrices as an array of real numbers is a primitive data structure in most high level programming languages.

Time Series

A time series, i.e. a signal represented by its values at different points in time is a very common object in control engineering and other disciplines. It may be the result of measurement on the real world, the result of a simulation, or it may be the value of some function of time, e.g. sin(wt).

In Idpac almost every command will use a time series either as input or output. Also Synpac, wich is used for the design of control systems in the time domain is heavily dependant on generating and visualizing time series, simply because the time behaviour of the system is the design criterion.

Most theory and many practical methods assume that the distances in time between different points in the time series are equal. This makes it possible to condense the time information and include a single specification of the time increment. At each time instant, more than one signal is usually measured, or simulated. In many cases it is natural to store values from more than one measurement point into the same series, e.g. output #1, #2, ... etc. into series Y, and input #1, #2 .. etc. into series U.

All such signal values belonging to the same time instant could then be stored in a single row in a matrix (array) while values belonging to other time instances are stored in other rows. Thus the number of rows is equal to the number of time instances while the number of columns is the number of individual signals represented.

Thus apart from the information on time increment, the number of individual signals and the number of sample times, a set of related time series are conveniently treated and stored as an array, cf. Figure 5.1.

Frequency Responses

A frequency response consists of a sequence of complex numbers, i.e. amplitude and phase, each associated with a real number, the frequency at that point. By including the frequency value at each point, total freedom in frequency spacing is achieved. This may be of great value in cases where frequency responses are measured. A frequency response is conveniently stored as an array.

Loci

A locus is an array of complex numbers. These numbers are typically either matrix eigenvalues or polynomial zeroes. Of course this would as examples include system poles/zeroes.

For each set of values, there is a real variable wich is the current value of a varying parameter. A typical example would be the loop gain in a root locus analysis. This is also the main intended use of such an object; by plotting the complex numbers in the complex plane for each value of the parameter, a locus for eigenvalues / zeroes may be obtained. It is easily parametrized in the parameter variable. Again we see that the basic structure of a two dimensional array will accomodate this type of object.

Polynomials

Polynomials are common objects in program packages for use in automatic control. They could be the numerator and the denominator of a rational transfer function or operator polynomials in the description of a system on matrix polynomial form. The latter case is more demanding in that it requires the possibility of a polynomial with matrix coefficients. Such a polynomial matrix can in a natural way be regarded as a three dimensional array. The coefficient matrices are stored in the normal fashion using the first two dimensions, while the third dimension is used to distinguish the coefficients for the successive powers in the independent variable. The scalar case where coefficients are scalar quantities is included by letting the first two dimensions parameters be equal one.

5.3 Systems

Granted that we deal with program packages for automatic control applications, the notion of a system in the meaning of a dynamical system, to be controlled, analyzed, modelled etc, is of vital importance. We want to perform certain computations on such systems, so we have to represent them in a computer in some way, but above all, we have to know what we mean by the word "system". It must be decided what is meant by a system representation and when different representations should be considered to represent the same system. It must be decided which representations should be legal and what safety measures should be put in the program.

Let us regard the system as a "box" (black, misty or transparant) with some inputs and outputs, possibly classified in some way as command inputs, disturbance inputs, measured outputs and controlled outputs. See Figure 5.1. We assume that the operations of this box can be described by differential equations, linear or non-linear. The black box case then refers to the situation when the nature of the underlying equations is largely unknown, while the transparant box represents the opposite case. Naturally, the misty box case is when the system is described by equations of known form but with unknown parameters.

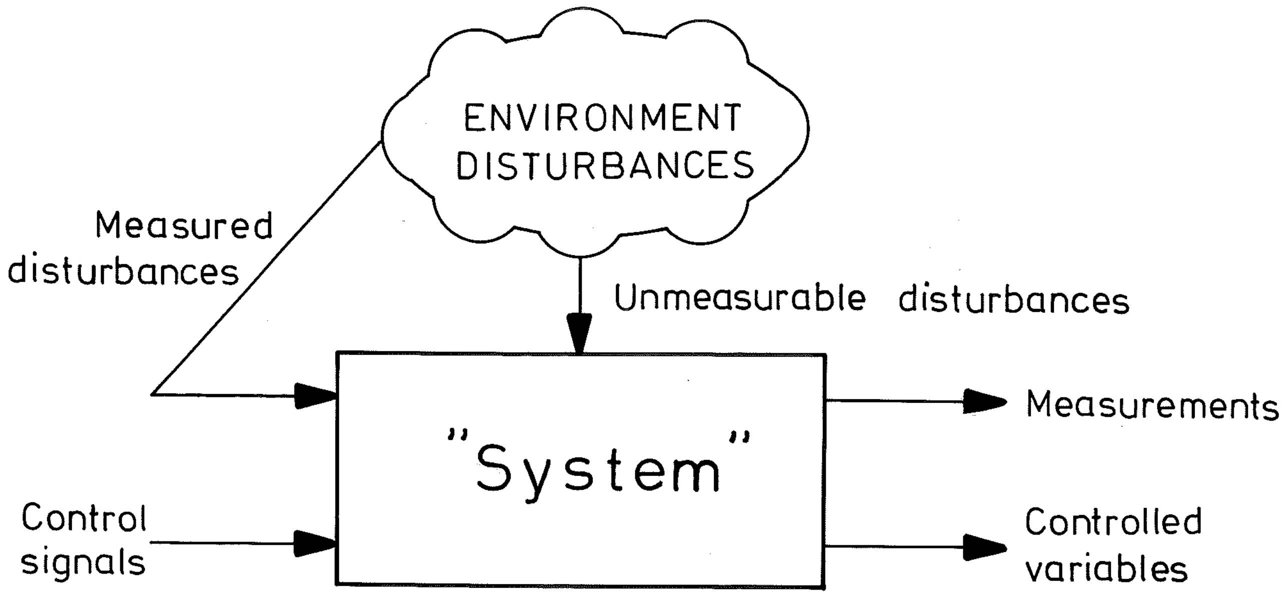


Figure 5.1. A general system with inputs and outputs.

A given system in this sense can be described in many different ways, depending on the insight we have in the box. One way is for example to record inputs and outputs and use these time responses as a system representation. If we know or are prepared to postulate a model structure there are different ways to compute estimates of parameters in such models. The models may be of different orders, giving different goodness of fit to the data. Also different model structures are possible: difference equations, state space differential equations etc.A transfer function description requires for example few assumptions other than linearity, giving the system characteristics as amplitude and phase curves.

If we approach the problem from the opposite direction, we may have sufficient knowledge of the system to be able to write down complete (high order) differential equations, based on fundamental laws in physics, chemistry, mechanics etc. At least linearized versions of such differential equations are in a natural way represented as (matrix) polynomials in the differentiation operator. Following certain restrictions, such a system can be transformed e.g. to state space form or transfer function form.

This discussion, more elaborated in Chapter 2, is repeated here to stress that several different ways of representing a given system is possible, and that all of them in some way describe the dynamical properties of that system. It is therefore strongly desirable that what is to be called a system in a program package includes the facility to allow simultaneous descriptions. On the other hand, a free inclusion of different system representations into a "system" is of course unnatural. There must be some restrictions, so that unrelated representations cannot, at least easily, be put together into one "system". Care must also be exercised when any representation is changed to ensure that all representations are equivalent.

Note that we already have tools at hand to deal with some aspects of the problem of storing a system description. Polynomials can store the coefficients of a difference equation while matrices are ideal to store coefficients of state space models. Frequency responses are available to store the transfer function as it is computed by spectral analysis. Time series can be used to store impulse & step responses and so on. Now, the ability to store sets of data is not enough, their interrelations must be given somewhere. An additional grasp on the subject is needed. Thus we find that the notion of a system leads us to the requirement that several different representations of a given system should be allowed. To the program package, they should appear as different instances of a "system object". The access to such an object must be subject to some conventions.

We could for instance demand that it would be possible to reference a system with a name, specifying the system as an abstract object, plus a representation name, specifying a representation from a set of available choices. The conventions should ensure that the representation forms available for a certain system indeed describe the same object. It should be legal to obtain a new representation through transformations of already available ones. It should not be possible to freely include new representations because of the risk that they might actually describe a "different" system. An example of such conventions is given later in Section 5.13. The conventions have to allow great flexibility, since many representation forms must be catered for. Other requirements include the possibility to get a quick overview of the system in its available representation forms. Coupled to this is a need for compactness. This need is apparent from a quick estimate of how many numbers are used even in a moderately complex model on e.g. state space form or transfer function form. Storage for all this information must be dynamically allocated to allow a great expansion potential.

Finally, it would be nice if the interactive user could start operations on a system or a representation of a system without having to specifically state the type of representation. As an example, a command of the type

```
SIMU Y < SYST(REPR) U
```

should be allowed (meaning simulation of the representation REPR of system SYST with input U and output Y) regardless of whether REPR happens to be a state space or a transfer function representation.

5.4 Review of Requirements

The requirements on data handling in an interactive program is by no means an elementary exercise in programming. We are required to handle objects referenced by the user via names. The naming mechanism is straightforward, but the objects are of different shapes and sizes.

Scalar objects were found in the form of iteration counts, parameters, dimensions etc.

Array objects were encountered as matrices and vectors, time series, frequency responses, loci etc. They pose no structural problem but they may be large. As an example, a time series may in a practical application contain 10 signals sampled at 1000 points.

System objects are the tricky part

System Objects

One problem is to allow several representations of a single "abstract system". This can be achieved by means of a suitable naming strategy. Another problem is to allow representations to be of different types. This can be solved by brute force in FORTRAN. It may be done more elegantly in high level programming languages, as is discussed in the next Section. The last and most demanding problem is that a system is a structured object, consisting of scalars, such as orders and dimensions, and arrays, such as matrices, polynomials etc. Also, some parts of a system representation may be optional, such as a noise input.

From practical considerations, there is a requirement that storage for these objects can be allocated dynamically. This should be feasible both for the total size of an object and for its internal representation during processing. The size problem can be exemplified by a state space system with A, B, and C matrices. It is quite conceivable that these matrices have to be stored temporarily inside a procedure (subroutine) during processing. If available temporary storage must be decided prior to startup of the interactive program, which is reasonable to assume, it is then also reasonable to demand that it is utilized efficently. This implies that if storage space corresponding to 675 real numbers were available, this space would be enough to store either a 15:th order system with 15 inputs and 15 outputs ($15^2 + 15^2 + 15^2 = 675$) or a 25:th order system with 1 input and 1 output ($25^2 + 25*1 +1*25 = 675$).

Finally, some operations in the interactive program is likely to be applicable to several kinds of objects. An example is simulation of a system which can be defined, although differently, for a state-space representation and a transfer function representation. It would be desirable from the user's point of view to be able to specify this operation without considering the actual representation form, and require the program to apply the correct algorithm. In other words, operations should be automatically typed according to the kind of objects they reference.

5.5 How It Could Be Done

This section will show a skeleton implementation of a Synpac like program package (synthesis of linear multivariable systems). It is written in a hypothetical high level programming language which leans heavily on the class concept of Simula [Birt73]. It also uses the type declaration and case statement of Pascal [Jens74]. A major new facility in this hypothetical language is that it is interactive. This will be commented upon at the end of this section. It should be emphasized that the examples are intended to illustrate some general ideas and are by no means complete.

Figure 5.2 shows the environment of the following figures. It shows the main idea of this section, that the objects such as signals, matrices etc., are naturally implemented as Simula classes. A class in Simula is a definition of a data structure together with operations applicable to that data structure. Several instances of a class may exist, referencable through reference type variables (pointers). Also, predefined concepts allow the objects to form sets. Moreover, Simula classes inherit properties in a controlled and natural way. As will be shown, this framework will make it easy to satisfy our requirements.

```
SIMSET begin
   type reptype = (SS,TF);
   type sigtype = (Step,Sine,Norm,Zero);
   class signal
      begin
      ...
      end;
   class matrix
      begin
      ...
      end;
   class polynomial
      begin
      ...
      end;
HEAD class system
      begin
      ...
      end;
LINK class representation
      begin
      ...
      end;
representation class statespace
      begin
      ...
      end;
representation class transferfunction
      begin
      ...
      end;
```

Figure 5.2. The overall structure of the Simula example.

Definition of Data Structures and Procedures.

The details of Figure 5.2 are then: first two scalar datatypes are defined; their use will be apparent later. Then three classes realizing signal, matrix and polynomial objects are defined, detailed in Fig. 5.3. Finally a family of classes realizing the system concept discussed previously are defined.

Figure 5.3 details the classes signal and matrix. As an example of their use, assume that the following declaration has been made:

```
ref(matrix) a,b,c;
```

and that a and b have been assigned suitable values. Then

```
c:-a.mult(b);
```

would compute the matrix product a*b and generate a new matrix-object, referenced by c, containing the result. Examples with signal objects will follow later.

The design of a set of classes depicting the properties of systems is more complicated. These basic properties are that a system may have several representations of different internal structure. By prefixing the entire data structure by 'SIMSET', a linking mechanism in Simula is made available. Then by prefixing the class system with 'HEAD' and the class representation with 'LINK', several representation objects may be linked together into the same system object. Predefined procedures like FIRST and SUC allow the referencing of connected representation objects, see Figures 5.4 and 5.5.

```
class signal(nsampl,nsig); integer nsampl,nsig;
   begin
   real array s(l:nsampl,l:nsig);
   procedure plot;
      begin
      ...
      end;
   procedure define(sig,typ); integer sig; sigtype typ;
      begin
      ...
      end;
   ...
   end;
class matrix(nrow,ncol); integer nrow,ncol;
   begin
   real array m(l:nrow,l:ncol);
   ref(matrix) procedure mult(b); ref(matrix) b;
      begin
      ...
      end;
   ...
   end;
```

Figure 5.3. Some details of the classes signal and matrix. The class polynomial would be very similar.

The actual definition of statespace objects and transfer function objects are done in Figures 5.6 and 5.7. Because they are prefixed by 'representation', they inherit the properties of that class, viz. the linkage into the system object and the representation name. The two classes define some procedures that were declared virtual in class representation. This mechanism allows the procedure body to be defined in its natural context, different for a statespace or transfer function object. A call to such a procedure, referenced by a ref(representation) variable, will automatically be routed to the correct version depending on whether the representation object is of type statespace or transfer function.

```
HEAD class system;
   begin
   ref(representation) procedure as(rname); text rname;
      comment This procedure searches a list of objects
              for an object with the name 'rname';
      begin
      Boolean ok;
      ref(representation) x,y;
      ok:-false;
      x:-FIRST;
      y:-NONE;
      while x=/= NONE and not ok do
         if x.name=rname then
            begin
            y:-x;
            ok:=true
            end
         else
            x:-X.SUC;
      as:-y
      end;
   procedure generate(t,n); reptype t; text n;
      begin
      ref(representation) x;
      CLEAR;
      case t of
         SS: X:-new statespace(n);
         TF: X:-new transferfunction(n);
         end;
      x.create
      end;
   end system;
```

Figure 5.4 The definition of the class system. The procedure as will be used to select a specified system representation.

```
LINK hidden class representation(name); value name;
                                         text name;
   begin
   virtual protected procedure create;
   virtual ref(signal) procedure simulate;
   virtual procedure transform
   end;
```

Figure 5.5 The definition of the class representation. Several represention objects may be linked into a system object (being a HEAD).

```
representation class statespace;
   begin
   ref(matrix) A,B,C;
   ref(signal) procedure simulate(u); ref(signal) u;
      begin
      ...
      end;
   procedure transformto(newtyp,newname);
                         reptyp newtyp; text newname;
      begin
      ref(representation) x;
      case newtyp of
         SS:begin
            x:-new statespace(newname);
            ...
            end;
         TF:begin
            x:-new transferfunction(newname);
            ...
            end
         end
      end
   hidden procedure create;
      begin
      ...
      end;
   INTO(this HEAD)
   end statespace;
```

Figure 5.6 The definition of class statespace.

How to Use the Data Structures

So far, we have encountered definitions of data structures and procedures that would implement operations similar to those of existing programs such as Synpac. We will now see how these tools can be used, cf Figure 5.8.

First, two identifiers are declared to reference signal objects, one identifier is declared to reference system objects. Then a new system is created, and the procedure generate is called (line 4). This procedure removes all previous representations if any (in case this was not a new system object) and then creates a new representation according to the desired representation type. The name of the representation is passed as an argument, and the body of

```
representation class transferfunction;
   begin
   ref(polynomial) P,Q;
   ref(signal) procedure simulate(u); ref(signal) u;
      begin
      ...
      end;
   procedure transformto(newtyp,newname)
                   reptype newtype; text newname;
      begin
      ref(representation) x;
      case newtyp of
         SS:begin
            x:- new statespace(newname)
            ...
            end;
         TF:begin
            x:-new transferfunction(newname);
            ...
             end;
         end;
      end;
   hidden procedure create;
      begin
      ...
      end;
   INTO(this HEAD)
   end transferfunction;
```

Figure 5.7 The definition of class transferfunction.

```
ref(signal) u,y;
ref(system) Adam;
Adam:-new system;
Adam.generate (SS,'Bertil');
Adam.as('Bertil').transformto(TF,'Caesar');
u:-new signal(100,1);
u.define(1.Step);
y:-Adam.as('Caesar').simulate (u);
y.plot
```

Figure 5.8 A simple example of the use of the earlier definitions.

the class statespace is executed. It consists of a single procedure call: INTO, which sets the linkage of this representation into the system object referenced by Adam. Finally the procedure create is called. Its procedure body is not shown but should contain actions to input the details of the desired representation.

Next, line 5, the list of available representations in Adam is scanned by the procedure as to find the one with name 'Bertil'. as returns a reference to that representation whose procedure transformto then creates a new representation of the desired type and name. The details of the transformation are not shown. Note that the new representation object will be linked into the system object, Which Adam points to.

A new signal is generated on line (6) and defined to be a step signal (7). With this step signal as argument to the procedure simulate (8), referenced through the transferfunction object with name 'Caesar', simulate returns a reference to a new signal object with the step response as value. The step response is displayed through the call to the procedure plot (9).

Special Features in the Hypothetical Language

Note that most of the skeleton Synpac shown in Figures 5.2 - 5.7 could be described in standard Simula with some imports from Pascal. One addition is the use of the keyword 'hidden'. It is used to prevent the use of classes or procedures from the user level. The procedure create is such a case. If the user could write

```
Adam.as('Bertil').create
```

he would be allowed to input new values into that representation, with the result that the system Adam would contain two completely unrelated representations (Bertil and Caesar). The keyword hidden is intended to prevent this.

Similarly the class representation itself is 'hidden'. This is to prevent the following construction:

```
ref(representation) R;
R:-Adam.as('Bertil');
```

This is not desireable because a system is the object to be seen from outside, not independent representations. The following construction would be really dangerous:

```
R.OUT;
R.INTO(Eve);
```

The effect of the two lines is to move a representation out of one system and into another. This ruins the intended integrity of system objects. The following two lines show how part of the data inside a representation is changed, also destroying the intended strict relation between representations of a single system.

```
R.A:-new matrix(2,2);
R.A.m[1,2]:=20;
```

These examples were made impossible if the keyword 'hidden' in

```
p hidden class c
```

were to imply that:

a) The user is forbidden to use ref(c) variables.
b) Data in the hidden class is not directly accessible.
c) Only procedures defined at the same or lower levels are accessible.

The most important addition to Simula used here is, however, that we assumed an interactive mode of operation. Unfortunately this is not possible in current high level programming languages. Otherwise, the design of an interactive program would have been possible using the same strategy used in many Simula programs. First the objects and procedures operating upon them are defined. This is a major task, Figures 5.2 - 5.7. Then the user, here in an interactive fashion, 'sets the wheels in motion' [Birt73], Fig. 5.8.

The next sections will show how these concepts actually have been implemented in FORTRAN. Structured objects are implemented as files, and keywords are used to distinguish between alternatives. It is less elegant and many things that was solved by the Simula language itself has to be done by program code.

5.6 Implementation in FORTRAN

There are one very important reason to choose FORTRAN as implementation language for an interactive program, viz. that no other programming language is generally available on medium sized computers. Yet, FORTRAN is inadequate for some functions that have to be performed, so the implementor will sometimes have to cheat, hopefully in a way that will not impair the portability of the program. The following problem areas will have to be solved.

a) Dynamic allocation of work areas.
FORTRAN does not provide for dynamic work areas (as Simula does). The use of variable dimensions in subroutine calls can partially solve this problem. An example is given in Figure 6.2.

b) Storage of, and operations on, non-standard values such as character strings.
FORTRAN has very limited data types. Operations on data of non-standard type can be introduced via calls to subroutines, maybe coded in a machine-dependent language. Allocation of and reference to such values will on the other hand have to go through standard data types. Compare the discussion in Section 6.5.

c) Dynamic storage of structured objects.
FORTRAN does not provide structured objects such as the ones in the Simula example, nor is dynamic allocation of program adressable data areas included. What is available on most medium-sized computers is, however, a dynamic allocation of files on mass memory. Although not standardized, the operations are very similar from computer to computer. This therefore provides a solution to the problem of dynamic storage.

The rest of this chapter describes the considerations that may be made using files as the vehicle for storage of structured objects.

5.7 Files in General

A file is an area on mass memory with some imposed structure. Like a data area in primary memory, it has some address which is used by the computer. For the user it is more natural to refer to the file/data area by e.g. a name. To accomplish this, the name is entered in a directory together with information on the file/data area. This information would be the hardware address, but could also include size, usage status etc. Most operating systems support mass memory files referenced by name. A directory is then maintained by the system programs outside the interactive package. The Lund programs depend heavily on such a feature because nearly all problem dependent data resides on mass memory and the names occuring in the command arguments are actually filenames.

Files may be of different types, sequential vs. random access and formatted vs. unformatted, to use FORTRAN terminology. A sequential access file can only be read or written sequentially from top to bottom. Random access files on the other hand offer the possibility to read and write values in the file in arbitrary order. This could be advantageous in a number of instances. An example in the Lund programs is the command PLMAG in Idpac where a short section of a datafile is plotted and individual data points may be altered. Another one is in the plotting command PLOT, where all data points have to be read twice, once to compute scaling coefficients and once to perform the actual plotting. In this case random access would allow less overhead, and the scaling information could be recorded for future use.

A severe problem with random access files, however, is that they must contain records of constant length known in advance. As we will see, the possibility to start reading a file without knowing details on its contents is of great value, and as sequential access files are quite adequate in most cases, such files have been used exclusively in the Lund programs. Sequential access files also have the advantage that they seem to be implemented in the same way on most computers.

The other distinction was between formatted and unformatted, or synonymously, symbolic and binary, files. Binary files contain information in the internal representation of the computer, thus requiring little overhead and offering compact storage. Symbolic files on the other hand contain information in an external representation, i.e. in e.g. ASCII or EBCDIC form. This means that data may be transferred to other computers or can be directly sent to a printing device. Such files can be checked, changed or generated virtually without restrictions using an ordinary text editor.

Symbolic files have the nice feature that they allow information to be associated with keywords. This means a great freedom when organizing such files, since no strict positioning rules have to be followed, i.e. "free format". Symbolic files are therefore advantageous where direct human interaction with the files is common, or when the files will be of vastly different types or structures. Such files are e.g. the system files in the Lund programs.

5.8 Data Files and the File Head

Data files is the common name of files of binary form in the Lund programs. Here some general considerations will be given.

To be able to read a binary file, one has to know in detail how it was written. This is because each WRITE statement in FORTRAN generates a logical record, while each READ statement may read more than one, namely if its I/O-list is longer than the record being read *). Thus WRITE statements and subsequent READ statements must agree. This is in many cases no restriction. Take as an example a program that

*) Ekman - Eriksson: Programmering i standard FORTRAN. Studentlitteratur, Lund 1973, page 78.

handles dynamical systems in, say, SISO transfer function form. Then the system would internally be represented mainly by two integers, the degrees of the numerator and the denominator polynomials, and the coefficients of those polynomials. Now, if we want to save this system on mass memory, we could do with a single WRITE statement:

```
WRITE (IDEV) NN,ND,(CNUM(I),I=1,NN),CDEN(J),J=1,ND)
```

This statement will produce a single logical record. When we want to restore the saved system, this could (and must) be done with a single READ statement with an identical I/O list. Simple as it is, this method has some drawbacks:

a) There will be a lot of files around with different internal structures. Attemts to read a file with wrong internal structure will almost certainly be catastrophic. The only way to avoid this is a naming convention, which reduces the freedom offered by user defined file names.

b) When program packages tend to grow or even multiply, they sometimes tend to be improved in other respects. Some of these are likely to bring changes in the internal representation of the objects that are handled by that program. This will also be reflected in the file contents and the programs must thus be changed in many places. If several programs should communicate with one another, changes must be made in all such programs.

c) Particularly in research work but also in general practice, it may happen that some operations of interest are not available in a ready made program package. In such a case, it would be desirable to be able to use the package up to the point where the special operation was needed and then to leave the package, perform the operation on data saved on mass memory and then return to the package and continue. In order to write the program to perform the special operation, detailed knowledge of the file organization of both input and output data types is needed.

The remedy to these problems is to introduce a flexible standardized file structure and try to stick to it. In the Lund programs, this has been done in the following manner. Files, jointly termed as data files and implemented as binary (unformatted) ditto, contain a first record of fixed length, the file head. The file head specifies the number of records to follow, and their lengths which are constant within a file. Figure 5.9 shows a detailed description of the file head. I1, I2 and I3 describes the structure of the file. Note that I2 specifies the record length, thus it is possible to correctly read the file, once the file head has been read and decoded.

Other information in the file head is the sampling interval wich is relevant if the file contains either measurement data or parameters etc. of a discrete time model. The date & time information in integers 5 & 6 is valuable as they give an identity to measurements and to results derived from them. The 7:th integer serves as an "escape" function as it allows non standard files and provides a means to stop reading such a file before any harm is done. The 8:th integer is a "fingerprint" in the sense that all commands

1	I1 (number of rows)
2	I2 (number of columns)
3	I3 (time dimension)
4	Sampling interval in time units
5	Date recorded
6	Time recorded
7	If zero, the record length is constant
8	"Fingerprint" (number of the generating command)
9	File type
10	Skip count

Figure 5.9. The file head format.

that generates a file puts its command number there. This can be used to check compatibility requirements. It may also serve as a debugging aid.

The 9:th integer is used to indicate the logical contents of a file. Examples of where and why this is useful is given in the special sections on data files etc. below.

The 10:th integer, finally, specifies a skip count, i.e. a number of records to be passed before the file can be treated in the standard fashion. This too is a kind of escape facility and is primarily intended as a way to extend the file head. In Simnon [Elmq75] this is used to indicate 9variable names and associated system names in a STORE file, to allow reference by name to the variables in a subsequent SHOW command. Conceivably, this extended file head could be used in a time series file to include scaling information, other statistical data, variable names etc.

5.9 Access Rules for Data files

There has to be some rules for controlling the use of data files. A few examples will illustrate this need. The examples show desired action types, but they also reflect the basic command philosophy of programs like Idpac and Synpac. Assume that DATOP is a command taking as input a column of the input file and that the output will be a column in the output file. The input and output could bethought of as a time series. Thus a simple example would be:

a) DATOP OUTFIL < INFIL (3)

Here column number 3 (i.e. signal #3) in INFIL is read, operated upon, and the result is a new single column output file OUTFIL. We have met the first rule:

1) If an output file name but no column number is given, a new file is generated. Any old file with the same name as the new output file is lost.

If on the other hand we want to keep old information in an existing output file, we can do so:

b) DATOP OUTFIL (2) < INFIL (3)

In this case only column 2 in the output file is changed while all others are kept as before. As all files are accessed sequentially, this implies that the old version of the output file is read and copied to a new file with modifications made to, in this case, column 2. This is governed by the second rule which reads:

2) If an output file with a column number is given, the new column must replace an old column or be number N+1 if N is the number of previously existing columns.

There is a shorthand description for the case where the input and output files have the same name:

c) DATOP < INFIL (3)

The rule describing this case is:

3) If the output file name is omitted, the input file name is assumed. If a column number is given for the input file, it is also used for the output.

It should be noted that these general rules have to be implemented in the command decoding part of the command modules. They may be augmented by other rules, specific for a single command or for a group of commands.

Note also that these rules stem from a set of desirable properties rather than from imposed properties of an operating system. Indeed, these rules will violate some limits posed by many existing operating systems. The seemingly innocent first rule states that it is legal to generate a new file with an already existing name, and that the old file should be automatically deleted (made inaccessible). Although this operation shouldn't lead to any problems, it may be illegal in some operating systems. The solution is then to demand the file handling interface to use a temporary name for the output, and then, in the closing operation on the output file, explicitly delete the old file and rename the temporary file.

It is more understandable that the operation to simultaneously read the old file and write a new file with the same name (example b & c) may cause the operating system to hesitate. If the files are private or if the system is single user, it should suffice to demand that the input is closed before the output. Anyway, the (transparant) use of temporary filenames will solve also this difficulty.

Summarizing, the rules 1,2 & 3 are natural to allow some desirable operations. If they are incompatible with the operating system, the file handling interface can be made to simulate the missing functions.

5.1Ø Specific Examples of Data Files

Some specific details on how the data objects of Section 5.2 actually are stored in data files within the Lund programs will be given here.

Time Series

We refer here to Figure 5.1Ø for a description of the file organization and the file head. A time series file is implemented as a standard data file with I1 equal to the number of time points, I2 equal to the number of measured signals (and the record length) and I3=1.

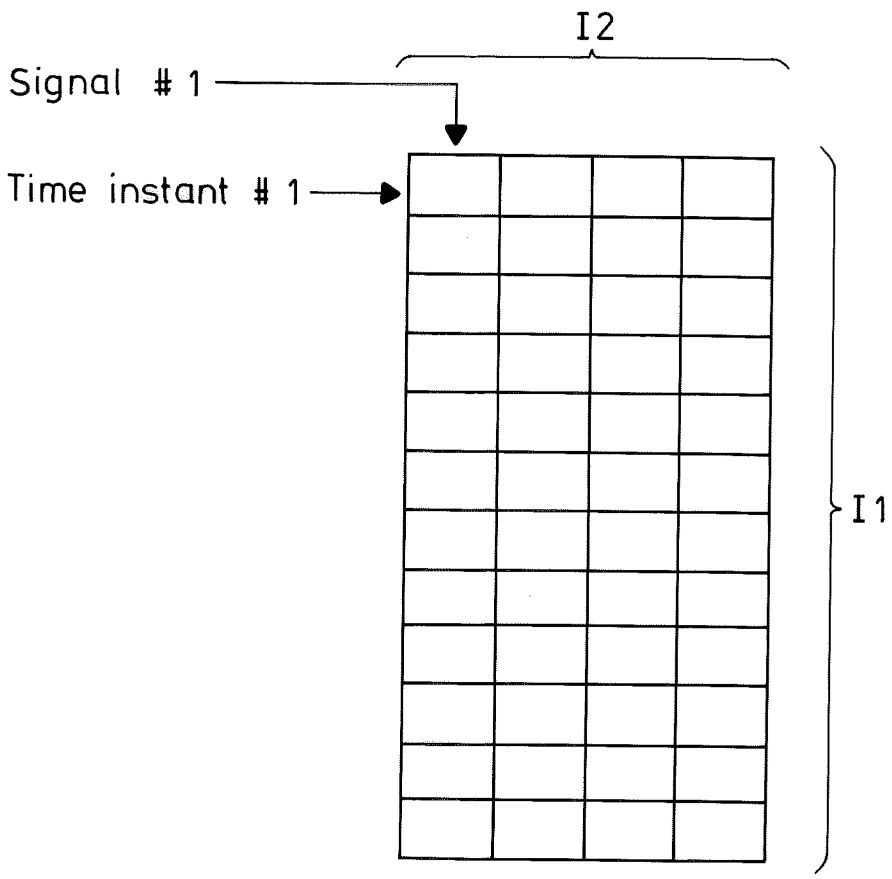


Figure 5.1Ø. The format of a time series file.

As indicated earlier, such a file will be produced in the Lund programs when performing a simulation or generating a function of time (i.e. using command INSI). There is also a logging program available, which generates a time series file. Here we encounter the reason for including the integer 7 in the file head. Such a logging facility is also provided in Simnon. In the logging program, it is possible to include a regulator to compute the input to the system, and it is also possible to have different sampling rate in the control loops. It is natural to put the contol signals into the log only at the points in time when they change. This implies that the signals will have different sampling rates. In this situation, our data logging program will use a non standard file format with non constant record length. Still the file contains the standardized file head with integer 7 flagging the special nature of the file. In such a case the file has to be treated and reeorganized in a special program before the data can be analyzed with programs like Idpac which assumes that the data has constant sampling period.

Matrices and Vectors

Matrices and vectors are easily stored within a data file. It is natural to store a matrix row wise, i.e. I2 (cf. Figure 5.9) is the number of columns in the same way as for time series. The number of rows is stored in I1. A vector is stored as a n*1 or 1*n matrix.

The time dimension I3 is used as a way to store time varying matrices (vectors). For each time instant (sampled data systems), the matrix is stored as above. The number of different time points is in I3. In the time invariant case I3=1. A 3*2 matrix is then stored with I1=3, I2=2 and I3=1. If it were time variable, it would be stored as I1=3, I2=2 and I3=100, assuming one hundred different time points were available. Compare this with a time response file with 3 signals: I1=100, I2=3 and I3=1.

Note that the method described here is influenced by history. A more natural way might have been to reserve one index, e.g. the 3:rd, for time information. This would leave the matrix method unchanged but the time series would be stored as I1=1, I2=3 and I3=100.

Frequency Responses

A frequency response could be nicely stored in three columns of a two dimensional array. The frequency values can be stored in the first column and the amplitude and phase in the 2:nd and 3:rd.

The main feature wich distinguishes a frequency response file from a time series file, is that data is recorded in groups of three columns. Some commands (like FROP, BODE and ASPEC, see Appendix) should recognize this and therefore this kind of file has a file code (integer 9) of its own.

Loci

The file format for a locus file is that of a data file with the parameter value in the first column. The complex eigenvalues / polynomial zeroes are stored with their real and imaginary parts in the following 2n-1, 2n columns, n=1,2 ....

Polynomials

A polynomial matrix can naturally be represented in the same manner as a time varying one, if the matrix coefficients for various powers of the independant variable is stored in the same way as matrices for various points in time. Thus a 2*3 polynomial matrix of degree 3 is stored with I1=3, I2=2 and I3=4, I3 being the number of coefficients.

Note that a scalar polynomial of degree N is stored as I1=1, I2=1 and I3=N+1. According to convention, the highest degree coefficient is always included explicitly and is stored first in the file.

5.11 Aggregates

So far we have treated data with a simple structure, stored in binary form. We have assumed that each such data set is interesting in itself. It thus makes good sense to store them separately, each in a single file. In many cases, such data sets are natural outputs or inputs from/to program modules, or are generated or inspected by such modules.

In other cases, however, the data sets are but different aspects of a greater entity. In automatic control, examples are descriptions of systems, see Section 5.12. In a complete description of a system on state space form according to Figure 5.12, eight or more matrices may be required and in the case of say a 5*4 system on transfer function form, the total number of polynomials in denominators and numerators would equal 40. After an initial phase when matrices or polynomials are entered and/or changed/corrected, they tend to lose their individual significance and are used only as parts in a greater scheme.

To store information on separate files would mean no conceptual difficulty, but would be a major practical one. On the computer systems where the Lund programs have been implemented so far, the time for opening, reading/writing and closing a single mass storage file, however short, is in the order of 1 second. A module to read in, modify and write back a system description on state space form might easily require 16 seconds only for file handling. In an interactive program package, this would lead to unacceptably long response times. The situation would be catastrophic even for moderately sized multivariable systems on transfer function form.

The solution to this problem is to introduce the notion of aggregate files. An aggregate file is the concatenation of several individual files of formats described above, into a single sequential file. Figure 5.11 illustrates an aggregate file. It consists of a file head with standard format. It is flagged as an aggregate by the file code being 100 in excess of the file code of the individual files it is made up of. The number of concatenated files is recorded in I1 which is also the record count for the records immediately following the aggregate file head. These records contain the file names of the constituent files. These files are then included sequentially in the order of their names, and are preceded by their file heads in the usual manner.

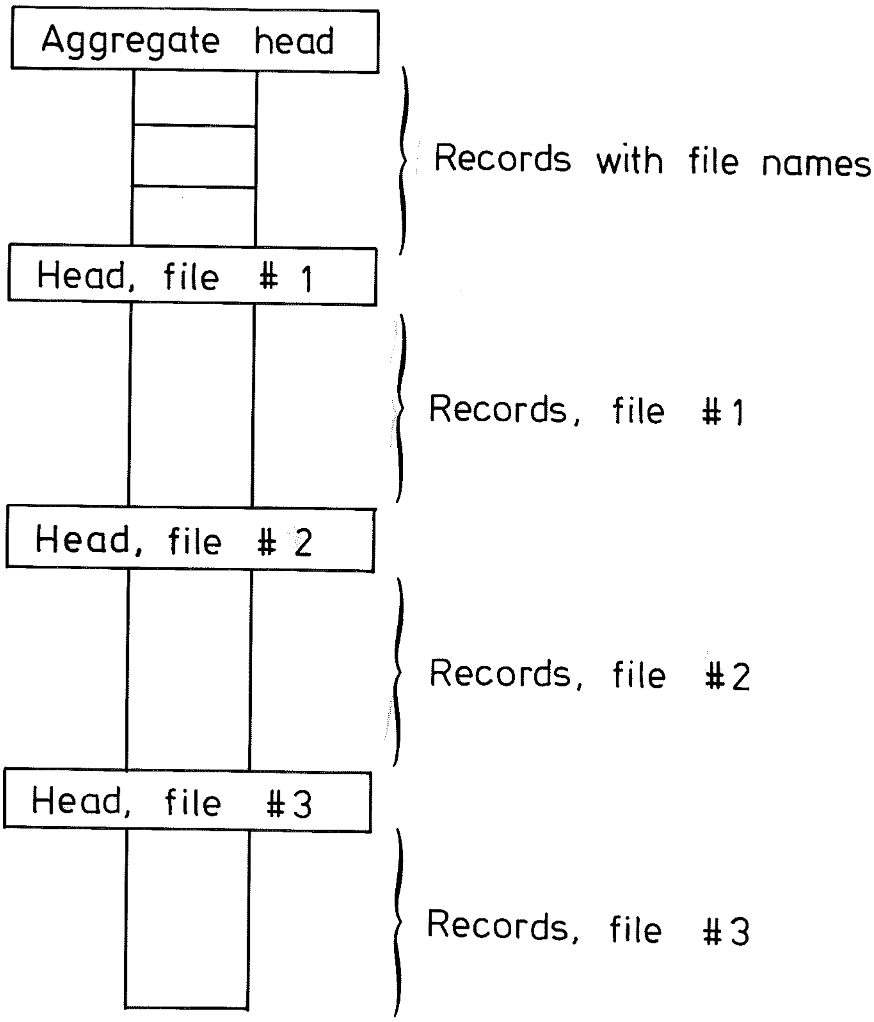


Figure 5.11. The format of an aggregate file.

The advantage of this scheme is that the file administration overhead in the computer system is paid for only once for a large set of related files. On the other hand, the advantage of the ability to handle these files separately, as in checking and modifying the data, is not lost. The program modules that do these operations can easily be made to allow a specified file to be a member of an aggregate. The time penalty for this is usually small since reading/writing past other files in the aggregate is a fast operation compared to the file administration.

To the benefit of the programmer, the file interface routines in the Lund programs are written to automatically handle files that are members of an aggregate file. The way this is achieved is by letting the file interface recognize and remember that an aggregate is opened. Open and close operations on that logical unit are then simply converted to operations that positions the file to the filehead of each successive file in the aggregate. Note that the file interface routines always have the possibility to know the format of the file and the position in it, due to the individual file heads.

Finally, note that although aggregate files serve as a means of increasing efficiency in accessing the data base, they also give the possibility of a nice naming convention. As an example, the matrices are not required to have distinct names any longer. They may be given standard names as found in literature such as A, B, C etc., functioning as 'forenames'. Only the name of their aggregate has to be distinct like a 'family name'. Examples on the use of aggregates are found in Chapter 7.

5.12 System Files

As was pointed out in Section 5.3, a system can be represented in a number of ways, with different types of data of varying structure, matrices, polynomials etc.

These facts speak for the use of text files. They give a greater freedom of structure since information is easily tagged with keywords and the recordlength is no major problem. Great amounts of data, e.g. matrix elements, polynomial coefficients etc., are stored in data files as described above; in the system file only the appropriate file names are given, sometimes within a "keyword structure", see the example in Figure 5.12. If matrices, polynomials etc are available as parts of aggregates, the aggregate filename is included. There are a number of standardized keywords specifying the type of system representation used. If there are more than one type of system representation present in the system file, they are enclosed within a pair of keywords BEGIN - END. Such sections within a system file are named separately, the name appearing after the BEGIN keyword.

A simple example of a system file is given in Figure 5.12. As we see, the system file contains a single section, delimited by a BEGIN - END pair. After BEGIN the section name appears, in this case 'cont'. Then the type of system representation is specified by a sequence of keywords: CONTINUOUS STATE SPACE REPRESENTATION. The initial state of the system is specified to be stored in a file with name xØvec. (In the example, filenames are written in lower case letters, keywords are in upper case).

```
BEGIN cont
CONTINUOUS STATE SPACE REPRESENTATION
INITIAL STATE VECTOR: xØvec
DYNAMICS, AGGREGATE: sysagg,
DX/DT= a*X + bu*U + bw*W + bv*V
    Y= c*X + du*U + dw*W + de*E
    Z= g*X + hu*U + hw*W
LOSSFUNCTION, AGGREGATE: qagg,
    QØ: qØ, Q1: q1, Q12: q12, Q2: q2
COVARIANCE FUNCTION, AGGREGATE: ragg,
    RØ: rØ, R1: r1, R12: r12, R2: r2
EXTENDED LOSSFUNCTION, AGGREGATE: qeagg,
    QEØ: qeØ, QE1: qe1, QE12: qe12, QE2: qe2,
    QE3: qe3, QE4: qe4, QE5: qe5
END
```

Figure 5.12. A section within a system file

After this, the state space system equations are specified following the keywords DYNAMICS and AGGREGATE. The aggregate name for the system matrices is given. The equations are written in full. They contain elements recognizable by the program such as DX/DT, *X, *U, Y= etc. These constructs serve to delimit filenames where the appropriate matrices are stored, a, bu etc. A matrix together with its key construct may be omitted and is then assumed to be zero, eg if dw*W is absent, then dw is assumed zero.

Optionally, a lossfunction, a covariance function and an extended loss function with respective aggregate names are specified. These things are used in the design of linear quadratic feedbacks (lossfunctions) or Kalman filters (covariance function). The use of the extended lossfunction is exemplified in Chapter 8.

In the case of lossfunctions etc. they are given in a form where standard keywords: Q1, R12, QE4 etc. are used. The definition of what one of these matrices actually stands for must be given separately in a User's Guide. One can ask why the same equationlike format that was used for the system equations is not used here. The reason is simply that such a representation would be more "messy", eg. the matrix q12 would have to be specified by

.... + xT*q12*u + ...

Here the keyword would be split into two: xT* and *u.

5.13 Access Rules for System Files

In the case of data files, the generation of a new file with a given name implies that any existing file with that name is lost and replaced by the new version. The reasons for this are explored in Section 5.9. In the case of a system file however, the corresponding situation is when a new section within an existing file is generated. Of course, the deletion of the entire old system file is out of question, it may contain much useful information in other sections. Therefore the old file is merely copied with the new section added at the beginning. This can only be allowed to happen when the operaton that generates the new section (system representation) is such that the new representation is just another way of representing the system.

Use as example the command FILT (in Idpac). Here a discrete time system with a certain transfer function is generated, to be used as a digital filter. Clearly it would be highly unnatural if it were possible to insert that filter description in a file describing a result of an identification experiment. A difficulty is that special cases always can be contrived. If for instance the filter in question was used to filter the raw data prior to the identification operation, it may be argued that the filter system in some way should be allowed to be connected to the resultant system file. On the other hand, if we are to stick to the basic rule of a system file containing different representations of the same system object, the inclusion of the filter cannot be justified. Note that it is possible for the user to include comments in the system file, being symbolic. Therefore, a note of the effect that filtering of the data prior to the identification has taken place, together with a reference to the filter system used, is quite feasible.

The access rules are as follows:

a) A command that generates a new system description will check that there is no already existing file with the output name.

b) If a command is used to transform system representations, a new section name must be specified if the output file name is the same as the input file name (or is omitted). If the output name differs, rule a) is applied.

c) A new section is placed first in the file. Any old section with that name is retained but may be accessed only through the text editor, rule d).

d) Only the first section with a given name is accessible through commands operating on system files.

5.14 Attributes

A system has a lot of properties which are more or less apparent from the system equations. Some of these, although of vital importance, are found very easily, but most often a rather complicated computation is needed. An interesting example is the degree of a system. This is trivially found by inspection for a system on state space form or on SISO transfer function form. For a system on MIMO transfer function form however, finding the degree of the system is a quite delicate problem.

These properties, characteristics or attributes are commonly computed and needed. It would be nice to have them stored somewhere, available for easy reference. There are two reasons for this, one being that it would give a valuable overview, the other being that in the cases where complicated computations were needed, they won't have to be done more than once.

Following the same arguments as for system files, we find that the attributes are most conveniently stored in a text file. It could be separate from the system file or it could be one or more sections, maybe with default names, within the system file. In either case there is a special problem that stems from the fact that some attributes will be invariant under transformations while others won't. An example is again the degree of a system. For a system on state space form the degree is unchanged for coordinate transformations but if we transform the system to transfer function form, the degree may change because only the controllable and observable modes are retained. Thus we have a situation where attributes are computed for a specific system representation but may or may not be valid for other representations of the same system, i.e. other sections in the same system file.

If we propose to store the attributes on a special file, we have to decide on the rules for this file. Two main principles are available. One would be to make the attribute file a mirror of the system file. It would contain sections each corresponding to a certain section of the system file, storing attributes for that system description rather than the description itself. When a new attribute was to be computed, all sections for wich the computation was valid were to be updated by the program, the rules governing the updating being known and possible to program.

This would be the ideal situation for the inexperienced user (cf. Section 3.9) who then doesn't have to bother with rules maybe unknown to him. On the other hand, the programming problems would be more difficult than they may seem at first sight. One is that in order for the program to decide if an attribute computed for one system description is valid for another, the entire history of transformations that has lead up to them must be analyzed. The transformations must therefore be stored. The other is: What should be done when a new section, i.e. system description, is added to the system file? Then a corresponding section in the attribute file would have to be generated, taking into account all previously available information and the relation of the new section to the previous ones.

A much simpler method is to let all command modules that compute attributes enter them into the attribute file under the section for which they are computed only. This would probably do for most cases, especially since a certain section is likely to be a "progenitor" for many others. If this is so, it would seem reasonable to use this section for computing all attributes applicable to the others. The method, however, does require the user to be aware of the rules and to be able to draw the proper conclusions regarding attributes for other descriptions himself. The user must therefore be the "experienced user" of Section 3.9.

Attributes are currently not implemented in the Lund programs, but are a considered expansion. The implementation as a file as discussed here is of course due to the assumption of FORTRAN as the implementation language. In the context of Section 5.5, the attributes would be naturally located in the system objects.

6. IMPLEMENTATION EXPERIENCES AND HARDWARE-SOFTWARE REQUIREMENTS

This chapter will try to give a rationale to some of the views given in the preceding chapters. The background will be a short historical account of the development of interactive programs at our department. This will serve as a framework to a discussion on choices made and requirements encountered.

6.1. The Start

One of the projects at the department in the period 1965-197Ø was to develop a library of subroutines for the numerical solution of many basic problems in control theory. Examples are solving linear equation systems, computation of eigenvalues, computing the matrix exponential, solving the stationary Riccati equation etc. This project was initiated by prof astrom who insisted that all software should be modular and conforming to certain programming rules.

The thing that started the work on interactive programs was the aquirement by the department of a "process control computer", ordered mid -69 and delivered mid -7Ø. The specification for the computer stated that FORTRAN should be available. A requirement for any other high-level language would have cut down possible choices to very few. FORTRAN was a natural choice also because of the fact that most of the numerical routines mentioned above were written in that language. We will see later that the choice of FORTRAN caused problems later on, but there were no real alternative at that time.

So called interactive programming languages were considered, mainly BASIC. It was investigated whether routines from a FORTRAN library could be called from programs written in BASIC. It was indeed found possible but only in a rather limited way. The language as such was considered too limited for general use.

The first projects done on the process computer included the development of a program for logging of data, and as an extension, a program for on-line identification. The projects resulted in the program LOGGER (Sture Lindahl) and an on-line identification program [Jons71] based on recursive least squares. The real time environment made interaction natural. These first programs were question & answer oriented.

6.2. The First Interactive Programs

The availability of computing power 'open shop' made interactive computing to solve common every-day problems of control system analysis and synthesis feasible and attractive. It turned out that the "process computer" was used rather heavily to run small "one-shot" programs.

The first project to develop a general purpose program for this type of problems never left the writing desk [Wies70]. It was, however, a good exercise. The objective was to evaluate matrix expressions, allowing inverses and matrix exponentials etc. The lesson learned was the importance of user-defined names for results and temporary variables. Expressions were written and immediately evaluated, therefore the program actually was command oriented.

The first full scale project was Synpac [Nove72]. This program was intended to implement basic linear quadratic design. It included operations to handle dynamic systems on state space form, as well as an algorithm to solve the stationary Riccati equation. Already this first version of Synpac featured a command line with many of the characteristics of today's programs. The command decoding was done in assembly language routines, a natural choice due to limitations in FORTRAN and a scarcity of core space. Equally natural was to decode arguments immediately and pass them on to the application routines as values. The prime concern was a centralized code to allow a flexible and free format input.

This early version included a macro facility in its simplest form; the input to the command decoder was switched to a mass memory file. No arguments were allowed, nor any "general purpose commands" as in Section 4.5. Similarly, systems (Section 5.12) were implemented simply as a file of file names. Several lessons were learned from Synpac. After the basic version was developed, there were intensive discussions among several different users concerning suitable commands and features of the program. This then lead to further extensions of the package.

A project to develop a program to aid in simulating non-linear differential equations was started as a direct consequence of the success with Synpac. The result was the first version of Simnon, [Elmq72], which later was expanded,[Elmq75]. In fact, what evolved as Simnon was actually concieved as a command in Synpac.

6.3. Interactive Hardware

The hardware used to interact with the computer and its programs was from the beginning a mechanical teletype (KSR35), and a storage tube oscilloscope. The Tektronix 4010-series display terminals which are now in common use were not yet advertised when the project was started. The generation of curves as well as text on the display oscilloscope was done entirely by software.

Later experience with more modern equipment, i.e. 4010-series graphical displays or a separate alphanumeric and graphic display, shows that the original setup was ideal. The drawbacks of the printer, its low speed and its noise, is made up by its paper copy of the data entered or received from the computer. The 4010-type display mixes figures and diagrams with the input to the computer and after a while, the display screen will be virtually unreadable, unless it is erased. Once the screen is erased, you have to rely on your own memory. Two separate screens, not very common, one for alphanumeric the other for graphic information is better, but not good. Also in this case, the past actions will soon scroll off the alphanumeric display.

The solution to this problem is to design the program to generate a log of past actions that any time may be displayed. Eventually the log would be output on a printing device. The importance of the log was emphasized in Section 3.6 h.

Recent hardware development, mainly low-cost semiconductor memories, has made possible display units with separate alphanumeric and graphic capability. The result is two separately scrollable displays, one containing past operations, the other text and figures output from the computer. This seems to be a satisfactory solution, although it demands some dexterity in handling the hardware on the part of the user. Still the problem of a paper copy remains.

Note that the use of the more expensive graphical displays with display processor and light-pen facility has not been treated here. Their fundamental way of operation is advantageous in other types of applications, viz. where the ability of the light-pen to point not only on the screen, but effectively directly into the display memory, is of prime importance. The predominant use of the display in the applications described in Chapter 2, is to output results in the form of diagrams etc. With few exceptions, the user's response is not to alter the data presented directly, they are but representations of the result. Rather, the user's action will be to apply another algorithm or to change parameters to obtain new results. They are again presented in graphical form, maybe also compared with the previous results.

The light-pen is sometimes used to implement the man-machine interaction to specify the desired actions to be taken. This "pressing of light-buttons" as it is sometimes called, can not be considered more expedient than to press keys on a keyboard.

6.4. Evolution (1973-1976)

The experiences with Synpac gave a taste for more. The on-line identification program already mentioned was the next to be considered. It was interactive, but question & answer oriented. Furthermore, it was limited with regard to available operations. It was now expanded with spectral analysis routines, maximum-likelihood identification routines and above all, it received the same set of interaction routines as was used in Synpac. The on-line capability was discarded, not being generally useful. The reason not to retain it was that identification algorithms usually are sensitive to things like bias or trends in the measurements. The removal of biases and trends, scaling of data etc., is thus an essential capability, cf. Sections 2.2 and 2.4. Therefore, identification is in practice an off-line operation, and Idpac, as the name of the new program was, was amended with operations to perform these tasks, [Gust73] and [Wies76].

Three tendencies became apparent during this period. The first was that commands were being designed as natural steps in a design or analysis process, rather than just reflecting the different parts of an algorithm solving the problem. This meant that the first idea of making subroutines in the subroutine library interactively available was abandoned. In parallel with this trend, data were being organized in a structured way (i.e. systems), not only as primitive data types like matrices and polynomials.

Secondly, the possibility of the macro facility became evident. Macro arguments and general purpose commands were included. As argument transfer by value was implemented in the command decoding, it was natural to retain this rule, although restrictive (Section 4.5).

Thirdly, as more effort were put into these programs, incentive to avoid their being prisoners on the process computer grew. In other words, portability became a main concern. This resulted in the interaction routines being rewritten into FORTRAN. They are then known as the subroutine package Intrac [Wies78]. Other areas of importance for portability are treated in the following section.

6.5. Software Problems

As mentioned in Section 6.1, FORTRAN was chosen as the implementation language. FORTRAN has two advantages: it is available on virtually all computers, and many implementations give efficient code for many kinds of algorithms for solving numeric problems.

The programmer has to cope with many drawbacks though. FORTRAN is extremely weak on nonnumeric problems due to its few primary data types and lack of structural elements, both for data and code. FORTRAN also shows strong influence by its origin as a language for batch-oriented operations. An example is the response to a run-time error in input data: immediate return to the operating system. The following is a list of problems encountered and how they were solved.

a) Differences in FORTRAN Implementations

Although programs are constructed in accordance to all known rules on standard FORTRAN, it happens that they compile and run without problem on one implementation, but fail in either respect on another. This seems to be due to a lack of precision in the language definition. The only way to solve this problem is to wead out the constructions that cause problems as they become known, sometimes causing much extra work and expense.

Different interactive programs from this project have been implemented on a number of computers. Examples of minicomputers that have been used are the PDP-15 where the initial parts of the project were implemented, PDP-11, Nova and HP 3000. Larger computer systems used are the DEC-10, UNIVAC 1108 (the home for the later parts of the project) and CDC Cyber. There has been problems of the type above, and sometimes due to a more restrictive implementation. A general problem on the mini computers is the smaller primary memory.

b) Character Strings

When decoding a command line, the input, received as a string of characters, is to be scanned, subdivided into items, and interpreted. These operations call for handling character data, not available in traditional FORTRAN, although many implementations allow nonstandard constructs for this purpose.

The approach taken was first to decide how to store character strings such as filenames, variable names, and flags. The objective was to accomodate computers with at least 16 bit wordlength. On such computers, a real datum will be stored in at least 32 bits. These 32 bits would in turn allow at least 4 characters to be stored in whatever internal representation for characters used on a specific computer, although this scheme is likely to waste some bits in many cases. The storage of character strings having a maximum length of eight characters thus requires two adjacent real variable locations.

The resulting rules thus specifies that whenever a data area or some variables are needed for operations on character strings, they are declared as real variables or real (2,.) arrays. This allow us to reference such objects in FORTRAN in a machine independent manner, and then by specifying that all actual operations be carried out inside a small set of subroutines, only those routines will be machine dependent. The precise definitions are found in [Esse77a].

The recently standardized FORTRAN 77 allows a primitive type 'character' and associated operations. This version of the language is not yet generally available.

c) Variables of Varying Types

The decoded command line arguments are stored together with relevant type information in a "vector", cf. Section 4.3. Due to the lack of structured variables in FORTRAN, this vector actually has to be implemented as a number of arrays; one of integer type to hold a code specifying the type of data in position I, another to hold a possible integer value, one of real type to hold a possible real value, and finally a two-dimensional array to hold character strings.

This somewhat clumsy method would not be necessary in a language like Pascal, which allows much more elegant constructions as is illustrated in Figure 6.1.

d) Problems of Unknown Size

In a program package like Synpac, it would be very unnatural to write the algorithmic subroutines to reflect a certain maximal problem size in the definition of temporary data areas, especially in the light of the comment made in Section 5.4 on systems. In Algol-like programming languages, like Algol itself and Simula, this is no problem since temporary arrays of any shape and size may be defined and passed as parameters to procedures. Unfortunately Pascal suffers a severe deficiency in this respect as the size is considered part of the type of an array variable, making this problem impossible to solve.

```
type argvectyp = record
                   case argtype: (int,re,str) of
                     int: (iarg:integer);
                     re:  (rarg:real);
                     str: (sarg:string);
                 end;
var argvec: array[1..n] of argvectyp;
```

Figure 6.1 The type definition and declaration of the argument vector as it could be done in Pascal.

In FORTRAN, many (but not all!) implementations allocate temporary storage internal to subroutines statically and require its size to be a compile time constant. The "variable dimension" feature applies only to formal arguments in subroutine/function definitions. The solution to the problem used in Synpac etc. is based on this feature, although it also involves a certain amount of "cheating". What is used is that all known FORTRAN compilers seem to allow an array element as actual parameter to correspond to an array formal parameter. The array element could be a suitably situated element of a vector used as a common resource of temporary storage. This 'allocation' vector would be used by several subroutines, which receive their temporary storage areas through their formal arguments and redefine them into suitable shape and size through a variable dimension declaration. Rules for this and other programming issues are found in [Elmq et al 76], where more complex examples than the one in Figure 6.2 are found.

```
SUBROUTINE SUBI(N,A,B,IA,IB,W)
DIMENSION W(1)
KW1=1
KW2=KW1+N
CALL SUB(N,A,B,IA,IB,W(KW1),W(KW2))
RETURN
END

SUBROUTINE SUB(N,A,B,IA,IB,W1,W2)
DIMENSION A(1A,1),B(IB,1)
DIMENSION W1(N,1),W2(N,1)
...
...
RETURN
END
```

Figure 6.2 Subroutine SUBI, which is the one seen by a subroutine library user, allocates temporary storage from the vector W, and calls SUB to do the actual job.

e) Passing References to Data Items Declared on a Lower Level

This is a problem that originated in Simnon, when a "FORTRAN system" ([Elmq75] Chapter 5) notifies the main section of the program that it contains variables that should be interactively accessible as 'parameters'. That is, they should be accessible through a suitable name in e.g. the commands PAR and DISP. This can not be done in traditional FORTRAN, since it is required to handle the address of a datum, called a pointer. The solution used in [Elmq75] is to include two assembly language procedures to fetch and deposit real variable values passed as arguments at an address contained in an integer valued argument. Again we fool the compiler with a machine dependent solution.

In languages that allow pointer variables there will not be any problems. The datum used as parameter will be allocated on the heap by a procedure normally called 'new' and handled through its associated pointer variable at both the low and main level.

f) Deficiencies in I/O

The output editing facility in FORTRAN, the FORMAT-statement, is obviously aimed at business or batch applications, where large amounts of data are to be output in tabular form, possibly on preprinted paper. For such applications, the FORMAT statement is powerful and adequate. For an interactive program, where the output occurs in smaller amounts, usually as a mixture of alphabetic and numeric text, the importance is rather to automatically achieve a neat output format of individual data, depending on their type and numeric size. It is a simple programming exercise to write such routines. However it can not be done in a machine independent way.

Also the operations to open and close mass memory files, a frequent type of operation, are poorly standardized, not even available in FORTRAN on some systems. For use in Idpac, Synpac etc., an internal standard has been developed [Esse77b]. This is largely based on the operations available on the PDP-15. It later turned out that this computer was unusually well equipped in this respect. However, with some exceptions the operating systems on other computers seem to offer the same type of capabilities, so it has been possible to rewrite this standardized interface to suit other environments.

g) Plotting Routines

Software to generate diagrams of various forms to be output on plotters or graphical display terminals is of course essential for interactive programs. Such software is available on a license or leasing basis from software firms or hardware vendors. Although the functional capabilities are very similar, differences do exist, and differences in hardware capabilities may be exaggerated rather than depreciated.

The approach taken to increase the portability of our programs was to ignore a possible use of features other than the basic operations of drawing lines and moving the 'pen'. These operations were to be performed in two subroutines MOVTO(X,Y) and LINETO(X,Y) which would be implementation dependent and should be written to utilize software hopefully already available on a specific host computer. Around these two routines, and two others used to output character strings and initialize, a complete plotting package was designed. Due to this very basic FORTRAN callable interface, the rest of the plotting package could be implemented entirely in standard FORTRAN. Thus the many routines generating graphic information in Idpac, Synpac, Polpac etc. use a self-contained and portable plotting package, documented in [Scho77].

h) Segmentation Software

The need for a nice program structure allowing easy segmentation was mentioned in Section 3.7. The power available on the PDP-15 in this respect was of great importance in the early stages. In fact, the lack of a proper segmentation program is one of the main causes of trouble found in moving e.g. Idpac to other minicomputers. The PDP-15 operating system was amended to allow random access to segments during execution [Wies73]. On most computer systems, segmentation has never caused any problems.

6.6. Maturation 1976-1979

During these years, the set of interactive programs reached a more stable state. Two new programs, Modpac and Polpac were also designed. These two fill some gaps between Idpac and Synpac (and Simnon). Modpac deals with models, i.e. it allows transformation and analysis of system representations, while Polpac is a package using algorithms for polynomials to solve design and analysis problems for systems on transfer function form.

During this period no great new inventions were made, rather the activity included correction of errors and implementation of new application facilities. Substantial portions of the programs were also rewritten to simplify their structure and to make them more portable.

This activity of consolidation was a natural consequence of both the earlier expansion and the experiences gained when exporting the programs to other computer systems.

6.7. Conclusion

Looking back on the project, now when most of the work has been done, it is evident that many details could or should have been done in a different way. The first observation is that the final dimensions of the project, both with respect to size and ability of the resulting programs, far exceed those originally anticipated. With today's knowledge, detailed specifications could be made and a substantial initial effort would be put into the design and implementation of suitable modules for use in the later stages of development. Among the old application routines in Synpac and Idpac, some have been revised or rewritten many times as conventions have been changed or common operations have been modularized.

The project grew organically. Many features available today were not originally planned. Rather, the possibilitites occurred as a result of using the programs on practical problems. Several design decisions were also based on the available hardware. All original work was made on a PDP-15 with 16k memory, later expanded to 32k. After 1975, more and more of the development work was moved to UNIVAC 1108, offering a more efficient environment to the programmers.

A very coarse estimate of the manpower put into the project ends at about 15 manyears. The effort required for the first implementations was comparatively moderate, in the order of 3 or 4 manyears. The work to make the programs portable and finding a set of suitable primitives was more time consuming, about 5 manyears. The rest of the time has been used actually designing new application routines. One should not forget the importance of the environment in wich the work has been done. It has in many cases been possible to use experience either in the form of good advice or asactual library routines. It is impossible to estimate this in manyears, but they would be many.

Programming Language

A question that could be asked is whether the choice of programming language would have been different 1979 than 1969, what are the candidates? New languages to be considered are Simula 67 and Pascal. Pascal can not be used due to the difficulty in passing matrices as arguments to procedures. This makes general purpose matrix routines impossible. The language Simula 67 is sufficiently powerful for our purposes. Some of the implementational characteristics might have looked quite different if Simula had been used because other solutions to some problems would have been possible, cf. Section 5.5. Note that the interaction sketched there needed not be used, an Intrac-type command dialogue could still be implemented, using only the data structure ideas of Section 5.5. The drawback with Simula 67 is that it seems to demand a rather powerful computer system. The language is not available on most mini- or midicomputers. If portability was a major criterion, the choice would probably again fall on FORTRAN. For this language also speaks the great number of numerical algorithms already available in FORTRAN.

Basic and APL

The final version of the interactive language implemented by Intrac, the command decoding module, bears at least some superficial resemblance to Basic. One could again ask the question if it would not be possible to use Basic as an environment to the application modules. There are considerations that still speak in favour of a new language although maybe similar to Basic:

a) For portability reasons, the interpreter should be written in a common high-level language.

b) Standard Basic contains constructs of no interest, such as READ & DATA statements.

c) A procedure call statement should be included.

d) The allowable forms of identifiers are not sufficient.

These incompatabilities are so serious that it is safest to take the decision at an early stage: It is <u>not</u> Basic we are interested in.

APL on the other hand is an extremely powerful language with many of the properties listed as desirable in the next subsection. APL has indeed been used to implement interactive programs, also for Automatic Control applications, as reported in [Shan77].

APL is a rather old programming language, [Iver62], and has for a long time only been available on big computer systems. It should not be denied however, that the failure of APL to become more widely used is also due to the peculiarities of the language; it is not like any other programming language! Two further comments on APL will be made:

APL uses a notation for expressions and procedure calls different from what might be called 'mathematical notation' used in most other programming languages. This means an additional effort for any new user which might be prohibitive for some of the categories of Section 3.9.

One of the main features of APL is its powerful set of operators or operations on operators. By using these features, complex data structures with associated operations could be constructed. Unfortunately, this power also involves a great danger in case of errors. If extensive checks have to be built into a program, the possibility of a short and elegant formulation of the operation of actual interest is of minor value.

Although it is hard to rule out the use of APL in a project to develop interactive programs of the type discussed here, the decision to use APL would be equally hard to take.

The Ideal Interactive Language

The task to briefly formulate some criteria on the ideal interactive language is of course difficult. Let us first emphasize that we are not interested in the situation where the user is a programmer who interactively designs a program or tests an algorithm. We are interested in the case when the user is actively solving application problems. The list of criteria for a langueage in this type of interactive use is then:

a) It should be a powerful general purpose language, similar in syntax and semantics to languages commonly available today. It should be efficiently applicable where FORTRAN, Algol, Simula etc. are being used.

b) There should be no distinction in the language on statements used in interactive or noninteractive mode.

c) It should be possible to call for the inclusion of a predefined set of declarations (types, structures, and procedures) at any time.

d) There should be no difference between preprogrammed facilities (as in c)) and user additions.

e) It should be possible to protect a specified set of procedures and data from direct use.

f) There should be a discernible distinction in access method for actual arguments in procedure calls as well as in procedure definitions.

g) It should be possible to draw on the huge amount of well tested numeric software written in FORTRAN available today.

The rationale for these demands is:

a), b), c), and d) defines a language that will behave and look like present day languages, and hence will be easy to understand and learn. The additional facilities enables the user to remain at the program main level where he/she can execute statements to call procedures, to incrementally include new procedures and to declare types or variables.

b) and c). Note that these two points demand incremental compilation.

e) recognizes that there must be one or several mechanisms for protection available. This was exemplified in Section 5.5.

f) tries to improve readability and the intuitive appeal of procedure calls. Many modern programming languages allow several procedure parameter passing methods; call by value, call by reference, copy on output etc. Astonishingly enough, no known language forces or even allows these choices to be visible in the call to the procedure, although it would improve readability considerably. In an interactive situation, where the user often will rely on his memory for the form of a procedure call, a possibility of a memorically and intuitively appealing procedure call form is of course especially important. In fact, this was one of the main objectives in the design of the currently available programs.

g) This a very natural demand, dictated by economic reasons. The earlier investments of money and manpower available in FORTRAN software libraries must be possible to use. This interest could be satified through a standardized call facility to separately compiled FORTRAN routines. Another solution would be a possibility to automatically translate FORTRAN code into a subset of the new language. Note that FORTRAN is quite adequate for numeric applications, so there is no real incentive to rewrite such algorithms other than for compatibility reasons.

6.8 The Future

Certainly, the future will see additions to the existing programs in the form of new commands. Some changes in the existing software could be discussed. Intrac could e.g. be made to allow a distinction between call by value arguments and call by reference arguments. This would allow the return of scalar values as results of commands, see Section 4.5. To be useful, such a change would require the redesign of the command syntax for a number of commands. It is doubtful if the benefit of this feature would be enough to warrant the amount of work needed. It will in any case not be done in the near future.

There are some points of a more general interest. One is the language problem. The new language, ADA, developed for the US Department of Defence seems very promising. According to the demands set forward bu the DoD committee (the 'Ironman' report), this new language would solve all problems mentioned in 6.5 except for plotting software. In a longer perpective, a revised implementation in this new language would seem natural. Still, the problem with numerical software written in FORTRAN would have to be solved and interaction would have to be built in through an Intrac like communication moldule.

The development on the hardware side will have much greater impact in the near future. There are two aspects of importance. One is the trend of the traditional minicomputer to grow both in computing power and in addressing capability, the latter coupled with virtual memory techniques. This will make the present somewhat arbitrary division of the available commands into several packages unneccessary. Also the implementation of all temporary datastructures as files could be abandoned. The file interface could simulate the file structure in virtual memory, making the hardware and memory paging system locate the data items, thereby gaining much speed.

The other development on the hardware side is the new generation of micro computers. They are characterized by comparatively high computing power and most importantly, a significant addressing capability. Together with cheaper and cheaper memory, they will make an implementation of Idpac, Simnon etc. in a desktop calculator economically quite feasible within a couple of years. This would bring about a revolution to the practising engineer and a great challenge to Automatic Control education.

7. EXAMPLE 1. IDENTIFICATION ON THE BALL AND BEAM PROCESS

As an example of the use of Idpac, an identification experiment on the ball and beam process will be described.

The process

The ball and beam process was designed and built shortly after the aquirement of a process computer to the department, cf. Section 6.1. It consists of a beam, rotated by a torque motor. On the beam, a steel ball is rolling in a groove. The principles are shown in Figure 7.1. The rotation of the beam is controlled through the voltage to the motor. The measured variables are the angle of rotation, φ in the figure, and the position, x, of the ball along the beam. The angle is measured in the standard fashion with a rotary potentiometer, while the position is measured using a linear potentiometer formed by the sides of the groove and the ball itself, see Figure 7.2 for a schematic diagram. Note the use of a capacitor and a high impedance voltage follower to reduce the effect of spotwise bad contact when the ball is rolling.

Expected process dynamics

The process is naturally divided into two subprocesses, the motor with beam, and the ball.

The dynamics from motor voltage to beam angle is determined by the electric characteristics of the motor and the inertial moment of the beam. A linear model would be:

$$G_1(s) = \frac{FI(s)}{U(s)} = \frac{K_1}{s(1+Ts)}$$

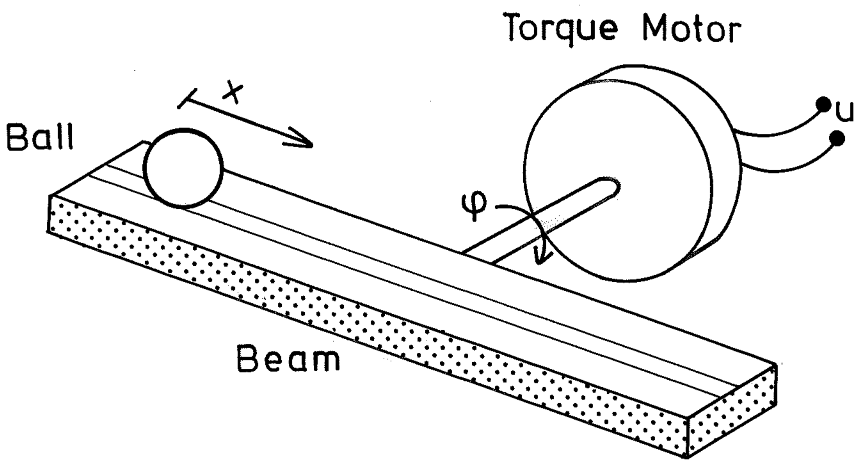


Figure 7.1. A schematic illustration of the ball and beam system.

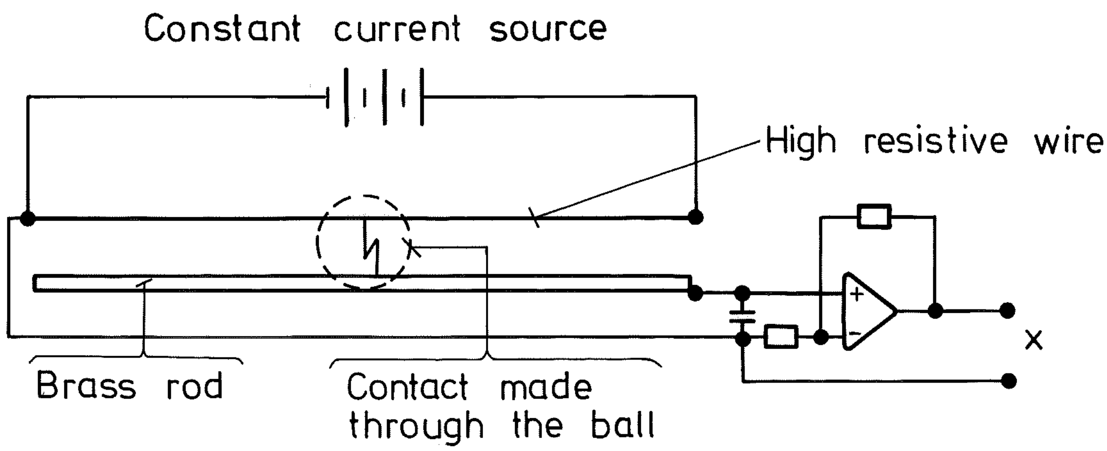


Figure 7.2. The measurement principle of the ball position.

This model is derived from the linear equations for the motor with inertia included assuming that the electrical time constant can be neglected. A torque balance for the rotor is:

$$J \frac{d^2\varphi}{dt^2} + \frac{C}{R_i} \frac{d\varphi}{dt} = \frac{k}{R_i} U.$$

J, the inertial moment is estimated from the geometry of the beam: 0.06 Nms^2. C, the induced voltage in the rotor is given: 0.56 Vs/rad. R_i, the internal resistance in the rotor: 4.9 ohm. k, the specific torque: 0.556 Nm/A. This gives

$$T = \frac{J R_i}{k C} = 0.95 \text{ s} \quad \text{and} \quad K_1 = \frac{1}{C} = 1.8 \text{ rad/Vs.}$$

The dynamics from beam angle to ball position is, if linearized, simply:

$$G_2(s) = \frac{X(s)}{FI(s)} = \frac{K_2}{s^2}$$

A constant angle will give a constant acceleration, hence the double integrator dynamics. When determining the gain, it is necessary to consider not only the inertia, but also the inertial torque and the rolling radius of the ball.

Figure 7.3 shows the notations used. We have

$$x = r\alpha \; ; \qquad \frac{dx}{dt} = r \frac{d\alpha}{dt} \; ; \qquad \frac{d^2x}{dt^2} = r \frac{d^2\alpha}{dt^2}$$

$$J \frac{d^2\alpha}{dt^2} = F r$$

$$m \frac{d^2x}{dt^2} = mg \sin \varphi - F = mg \sin \varphi - \frac{J}{r^2} \frac{d^2x}{dt^2}$$

With $J = m(2R^2/5)$ we finally have

$$K_2 = \frac{g}{1 + \frac{2}{5}\left(\frac{R}{r}\right)^2}.$$

For a ball with diameter 30 mm rolling in a groove with width 10 mm, K_2 gets the value 6.75 mr/s^2.

The experiment

A series of experiments on the ball and beam system was made. The problem was to excite the system without the ball falling off. The experiments were performed in the fall of 1975 and were partly inspired by the then current interest in the identifiability of closed loop systems [Gust et al74]. Two cascaded PD regulators were used in a configuration as shown in Figure 7.4. Several runs were made with varying parameter settings and varying perturbations. The experiment shown here is one with fairly good regulation and with the PRBS perturation as setpoint for the position

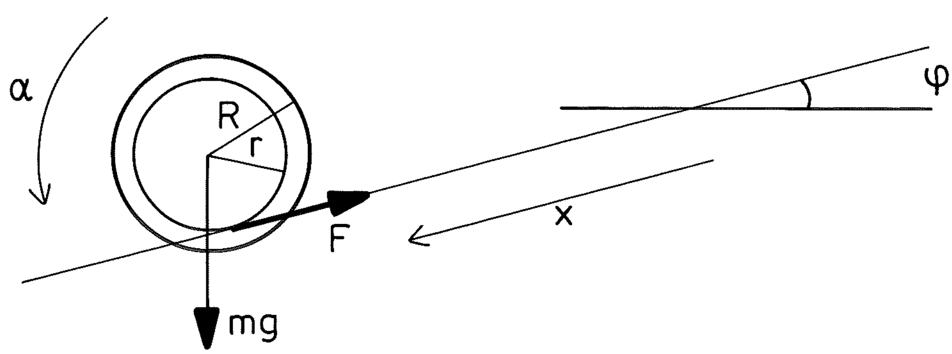


Figure 7.3. The symbols used in the discussion of the motion of the ball.

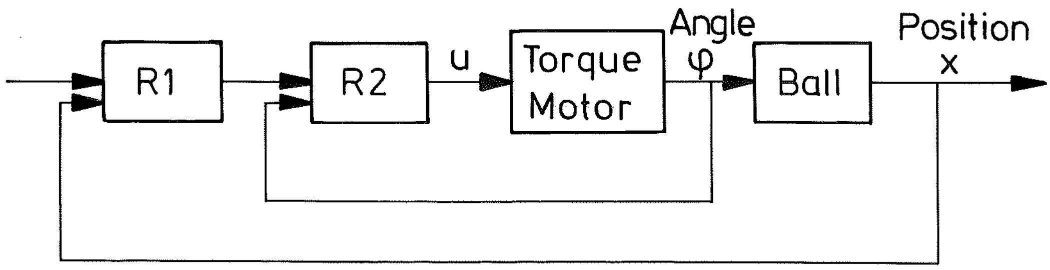


Figure 7.4. Block diagram of the control system for the ball and beam process.

of the ball. The sampling interval was short, 0.04 s. The experiment length was 700 samples, aquired during 28 seconds.

The regulation, the disturbance generation and the data recording was done with the data logging program mentioned in Section 6.1. Figure 7.5 shows the commands used to run an experiment in an interactive way. Note the use of guide lines. To each command there is a suggested successor although any command actually is legal.

The identification

The acquired data was later used for identification experiments using Idpac. The sequence of commands used for one of the runs is shown in Figure 7.6. A compact description of Idpac commands is included in the Appendix. On line 1 of the figure, the measurement data are converted to a standard time series file, and on lines 2-6, scaling and calibration operations are performed. After these operations X, the ball position, is in meters, FI, the beam angle, is in radians, and U the motor voltage, is in volts. The data thus obtained are plotted (lines 8 and 9) and the result is shown in Figures 7.7 and 7.8.

```
LOGGER V6A
COMMAND 'GUIDE ON' GUIDES YOU THROUGH THE LOGGER.
>GUIDE ON
COMMAND 'INFO' GIVES YOU INFORMATION ABOUT THE LOGGER ON LP.
>INFO
DEFINE  VARIABLES FOR  THE EXPERIMENT.  VARIABLE NAMES  WITH
DEFAULT ASSUMPTION  WITHIN PARENTHESIS:  COSAM(T), CORIT(T),
NVAI(2),  NVAO(1), NWRAI(Ø),  NWRAO(Ø), NWRUC(Ø),  NWRSL(Ø),
NUSAM(Ø), IH(Ø), IM(Ø), IT(Ø), NWTOT(Ø).
>NVAI<3
>NVAO<2
>NWRAI<3
>NWRAO<2
>NUSAM<7ØØ
>IT<2
>DONE
SET ANALOG INPUT MULTIPLEXER VECTOR (PRESET TO Ø,1,2,3....).
>DONE
SET ANALOG OUTPUT MULTIPLEXER VECTOR (PRESET TO Ø,1,2,3....).
>DONE
SET INPUT SCALE INDICES (DEFAULT Ø).
>DONE
IT MAY BE USEFUL TO SAVE THE EXPERIMENT CONDITIONS ON DT.
>SAVE BEAM
INPUT TASK NAME  FOR REGULATOR AND 'ON' IF  THE REGULATOR IS
NOT  TO BE  STOPPED  WHEN THE  EXPERIMENT  IS OVER.  DEFAULT
ASUMPTION 'NONE OFF'.
>REGNM IDREG
RUN THE  EXPERIMENT. OPTIONS  LP AND  DT FOR  CONVERT DURING
EXPERIMENT AND ALION IF ALIO  IS RUNNING (DEFAULT ASSUMPTION
NOT RUNNING).
>RUN BEAM1

LOGGER V6A
CONVERT DATA TO LP AND/OR DT.
>CONV < DT

LOGGER V6A
VERIFY EXPERIMENT.
>VERI
DOCUMENTATION OF EXPERIMENT.
>DONE
NEW EXPERIMENT.
DEFINE VARIABLES FOR THE EXPERIMENT.........
```

Figure 7.5. The interaction used to obtain data for the following identification. The guiding information hints at a possible next command, but it need not be followed.

```
1     >CONV ODATA < UNIDAT 5 Ø.Ø4
      >SCLOP FI < ODATA(1) - Ø.Ø781
      >SCLOP FI < FI / 2Ø.
      >SCLOP X < ODATA(2) / 2Ø.
5     >SCLOP U < ODATA(3) + Ø.13672
      >SCLOP PRBS < ODATA(4) * 1.ØØ4
      >LET NPLX. = 35Ø
      >PLOT FI X / (HP) PRBS(1) (HP) U
        >PAGE
1Ø    >CUT TX < X 34Ø 352
      >CUT TU < U 34Ø 352
      >STAT TX
      >STAT TU
      >CUT TX < X 44Ø 47Ø
15    >CUT TU < U 44Ø 47Ø
      >STAT TX
      >STAT TU
      >SCLOP T < X * 4.333
      >VECOP NU < U - T
2Ø    >PLOT U NU
        >PAGE
      >
```

Figure 7.6. The preliminary analysis of the measured data.

Observe that the input U does not have zero mean value when the ball position is constant. This is due to the disturbing torque from the ball, requiring a non-zero voltage to the motor for compensation. The least complex way to eliminate this effect, which would upset the identification results, is to use the measured ball position. This is (approximately) proportional to the disturbance. Lines 1Ø-17 show how the proportionality constant was found, while lines 18 and 19 is the actual removal of the disturbance. The new voltage signal NU is then used for the identification.

The identification procedure is shown in Figure 7.9. First, on lines 1 and 2, the input (FI) and the output (X) of the ball system are moved into a file WRK, whereupon the maximum likelihood identification command is used to obtain a model. The results from the first try indicate that some parameters are not significantly non-zero. (The complete output from the ML command is omitted here.) Therefore a second

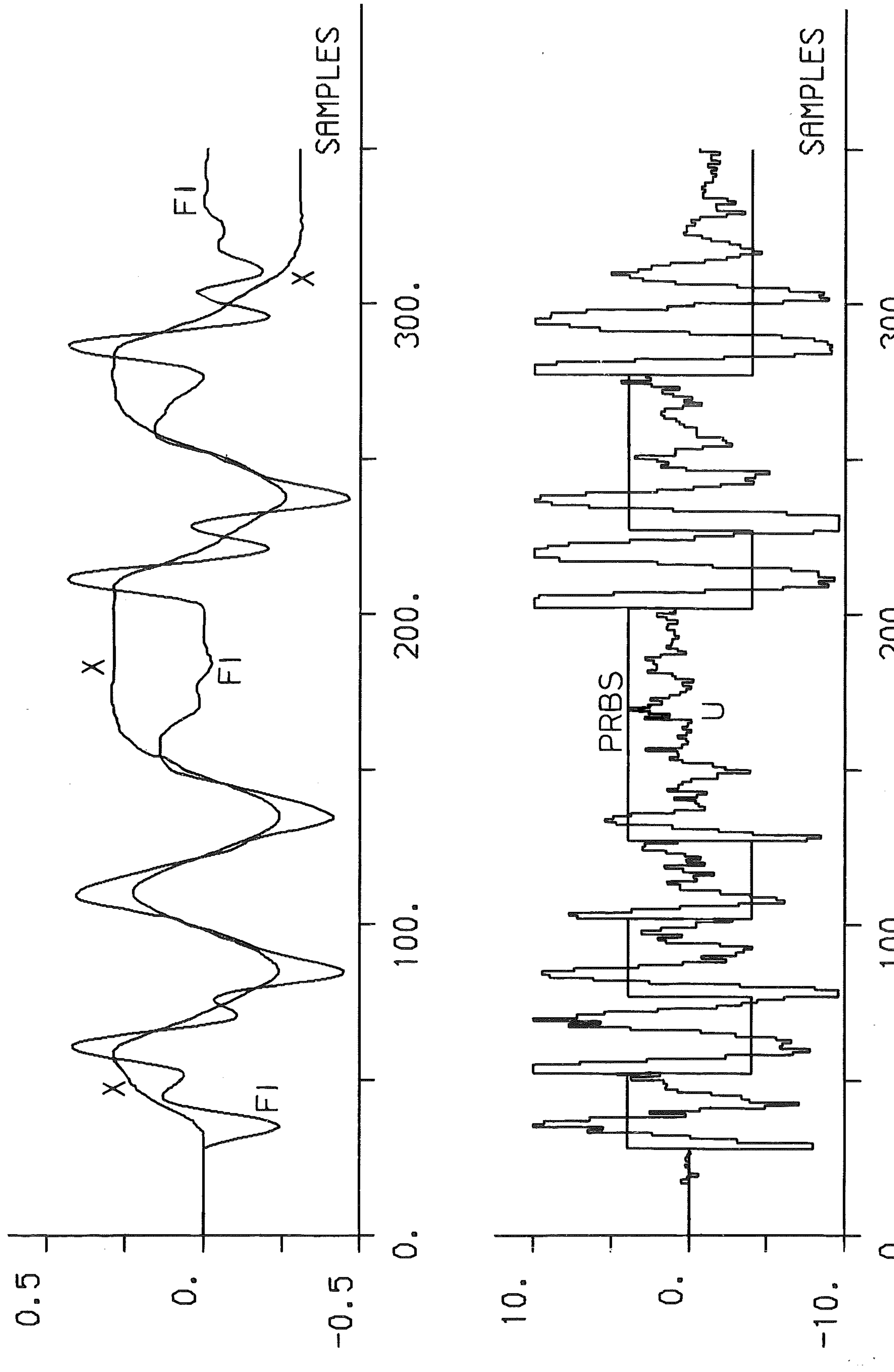


Figure 7.7. The beam angle (FI) and ball position (X) is recorded in the upper diagram, while the position setpoint (PRBS) and motor voltage U is shown in the lower diagram.

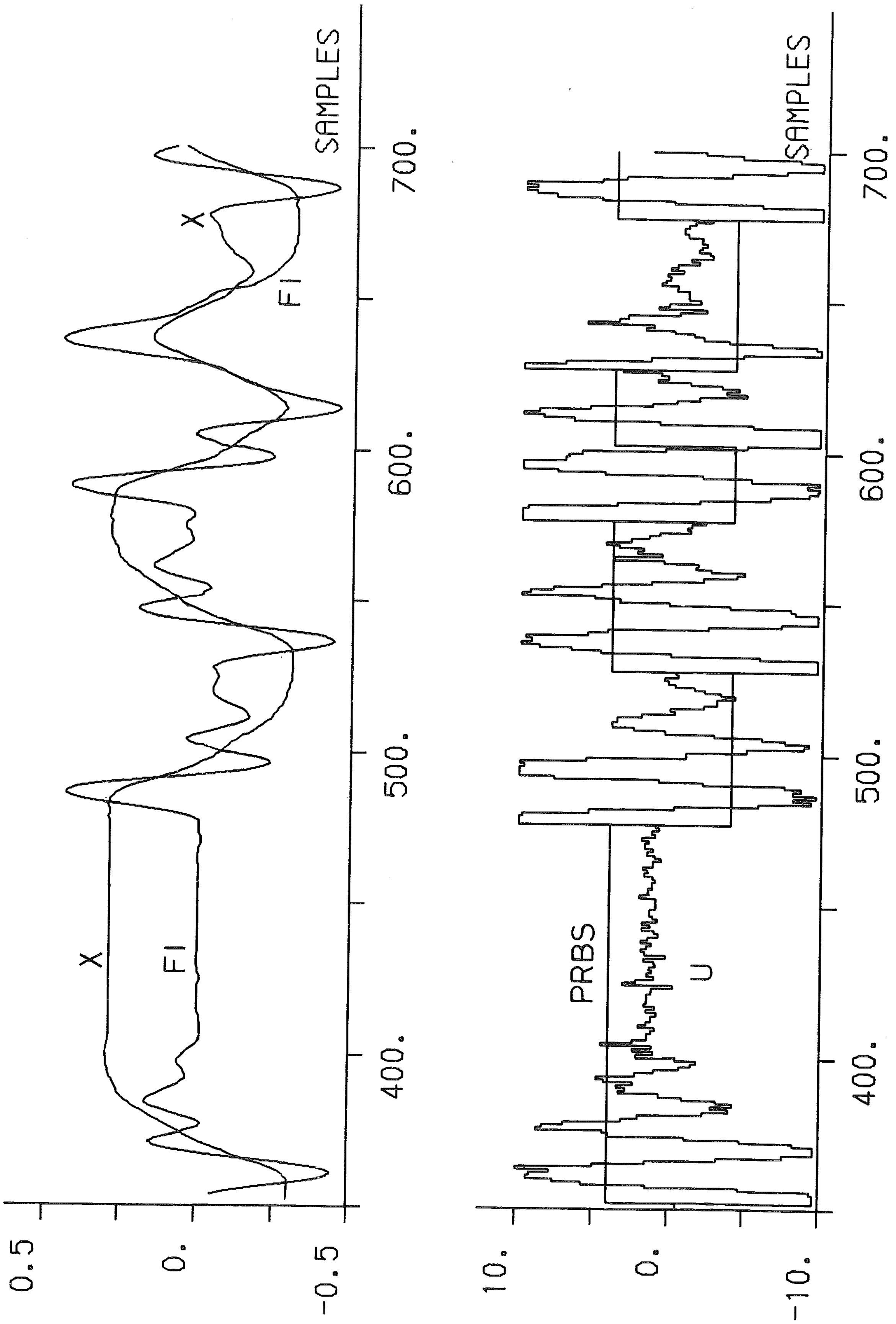


Figure 7.8. The continuation of Figure 7.7.

identification is performed (line 4) with the same starting values (line 5) but with some parameters fixed to zero. The resulting model is tested by a residual test, lines 10 and Figure 7.10, and through deterministic simulation, lines 12 and 13 and Figure 7.11.

The identification of the motor and beam dynamics is done in a quite similar fashion shown on lines 15-24. The residual test is shown on Figure 7.12 and the deterministic simulation on Figure 7.13.

```
1     >MOVE WRK(1) < FI
      >MOVE WRK(2) < X
      >ML MLBALL < WRK 2
      >ML (SC) MLBALL2 < WRK 2
5       >INVAL ABC MLBALL
        >SAVE STDEV
        >FIX B(2) 0.
        >FIX C(2) 0.
        >X
10    >RESID RB < MLBALL2 WRK
        >PAGE
      >DETER DX < MLBALL2 FI 280
      >PLOT FI / X DX
        >KILL
15    >MOVE WRK(1) < NU
      >MOVE WRK(2) < FI
      >ML (SC) MLMOTOR2 < WRK 2
        >SAVE STDEV
        >X
20    >RESID RM < MLMOTOR2 WRK
        >PAGE
      >DETER DFI < MLMOTOR2 350
      >PLOT FI DFI / (HP) NU
        >KILL
25    >
```

Figure 7.9. The Idpac commands used in the identification of the ball and motor dynamics.

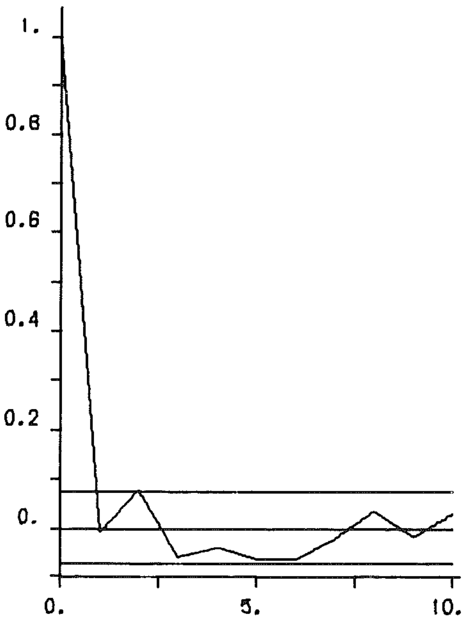


Figure 7.1Ø. Autocovariance function for the residuals in the ball model. If the function stays within the two lines, the residuals may be assumed to be independent, which they should be.

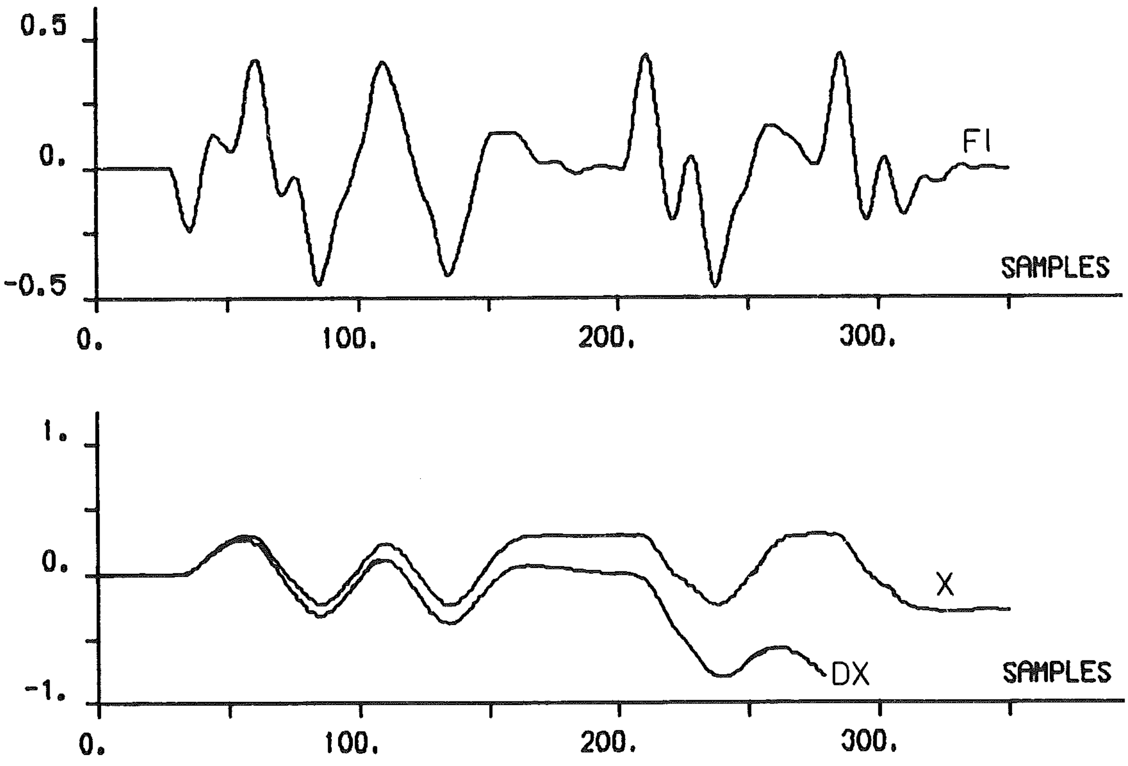


Figure 7.11. The input to the ball model (FI) and the measured position (X) and the deterministic output of the model (DX). There is a good agreement in high frequency behaviour, poor for low frequencies.

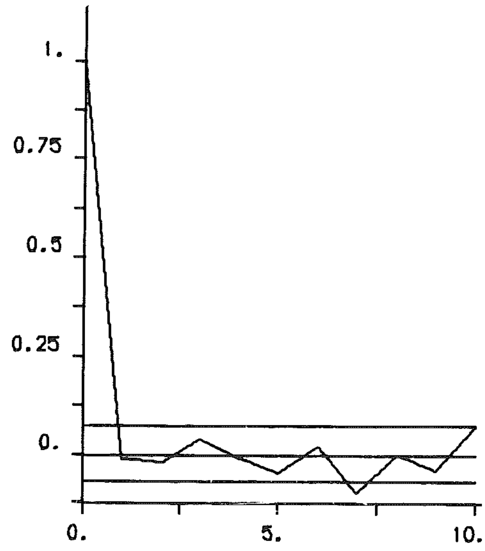


Figure 7.12. Autocovariance function for the residuals of the motor model.

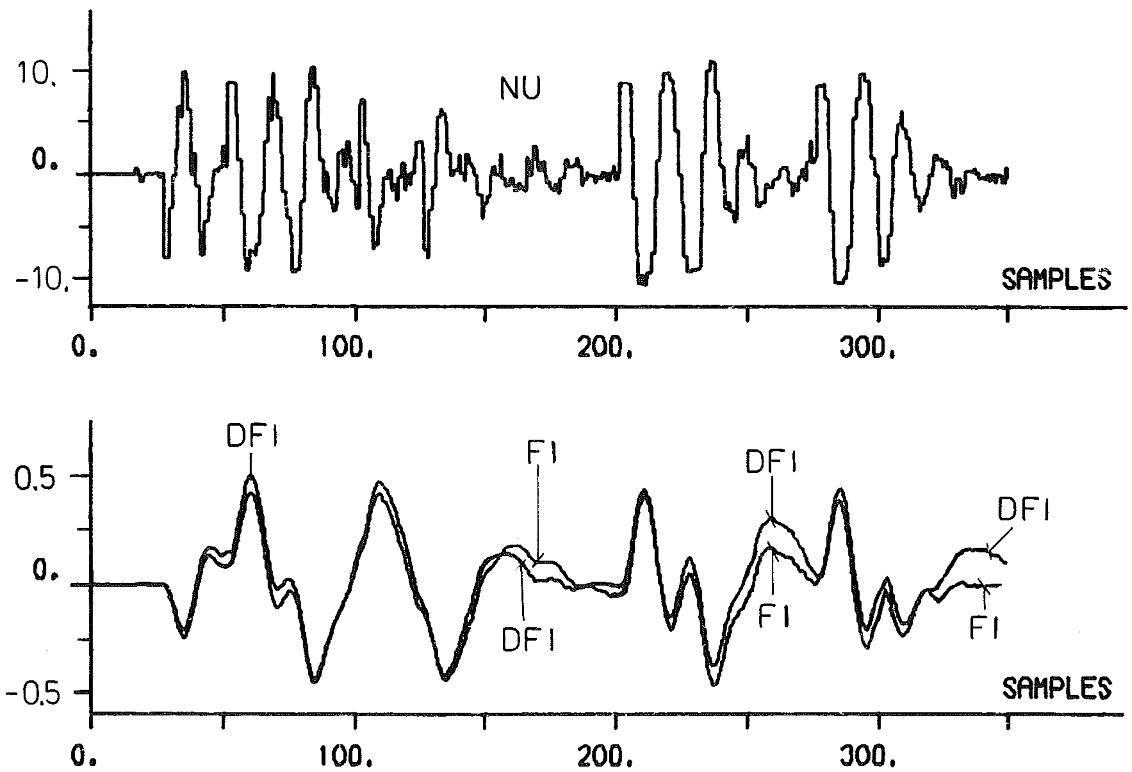


Figure 7.13. The input to the motor model (NU) and the measured angle (FI) and the deterministic output of the model (DFI). Again we have good agreement in high frequency behaviour, poorer for low frequencies.

Analysis of the models

The analysis of the models obtained in the previous section served to demonstrate that the identification procedure had been reasonably successful. The result will now be further discussed. First we are going to compute the system poles. Then the Bode diagrams of the obtained models will be compared with the theoretically expected curves.

The parameters in the 'Ay=Bu+Ce' model for the ball were:

a_1 = -1.9905 +- 1.9 E-3
a_2 = -0.98988 +- 1.9 E-3
b_1 = -1.05294 E-2 +- 2.3 E-4
b_2 = 0
c_1 = -0.695909 +- 3.2 E-2
c_2 = 0
λ = 0.00193

The corresponding parameters for the motor model were:

a_1 = -1.9775 +- 3.1 E-3
a_2 = 0.98341 +- 3.2 E-3
b_1 = 1.33906 E-3 +- 6.9 E-5
b_2 = 1.13159 E-3 +- 7.1 E-5
c_1 = -0.49411 +- 5.0 E-2
c_2 = -0.13521 +- 4.2 E-2
λ = 3.1784 E-3

Refer to Figure 7.14! First two systems are defined on line 1-6, using the Modpac command SYST. Each system consists of three sections (XDPOL, XDSS, and XCSS where X=B for the ball and X=M for the motor). The first section is a discrete time transfer function representation B/A. The other two are a discrete time and continuous time state space representation with A, B, and C matrices. Then the identified models are transferred to the corresponding system representation using polynomial files used in Modpac with POCONV, lines 7 and 8. Then the poles for the two discrete time models are computed, lines 9 and 10. Note the aggregate filename specification. The numerator polynomials of both representations are called A, but they are members of different aggregates. The default action of SYST is to name the aggregate file as the section name. The location of the poles is shown in Figures 7.15 and 7.16.

Finally, we are interested in the Bode diagram for the two models. First the models are converted into discrete time state space representations, lines 11 and 12, and then into the corresponding continuous time representations, lines 13 and 14. Finally the frequency responses for the two models are computed, lines 15 and 16.

Now, before we look at the result, the two theoretic models would be nice to have for comparison. To accomplish this within the framework of Modpac, two new systems are defined, lines 1-4 in Figure 7.17.

```
1   >SYST BALL(BDPOL) < (MTF) AB 0.04
2   >SYST BALL(BDSS) < (SS) ABC 0.04
3   >SYST BALL(BCSS) < (SS) ABC 0.
4   >SYST MOTOR(MDPOL) < (MTF) AB 0.04
5   >SYST MOTOR(MDSS) < (SS) ABC 0.04
6   >SYST MOTOR(MCSS) < (SS) ABC 0.
7   >POCONV BALL(BDPOL) < MLBALL2
8   >POCONV MOTOR(MDPOL) < MLMOTOR2
9   >POLZ BDPOL:A
10  >POLZ MDPOL:A
11  >TRFSS1 (BDSS) < BALL(BDPOL)
12  >TRFSS1 (MDSS) < MOTOR(MDPOL)
13  >CONT (BCSS) < BALL(BDSS)
14  >CONT (MCSS) < MOTOR(MDSS)
15  >SPSS BBODE < BALL(BCSS) 1 1
16  >SPSS MBODE < MOTOR(MCSS) 1 1
```

Figure 7.14. Modpac commands used in the first part of the analysis of the ball and motor models.

RE	IM
0.9699	0.0000
1.0206	0.0000

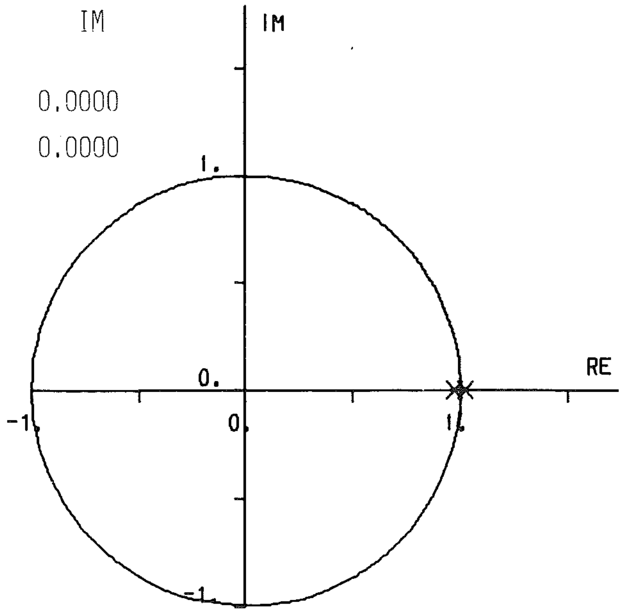


Figure 7.15. The pole configuration for the ball model. There is one unstable mode.

RE	IM
0.9888	7.6049 E-2
0.9888	-7.6049 E-2

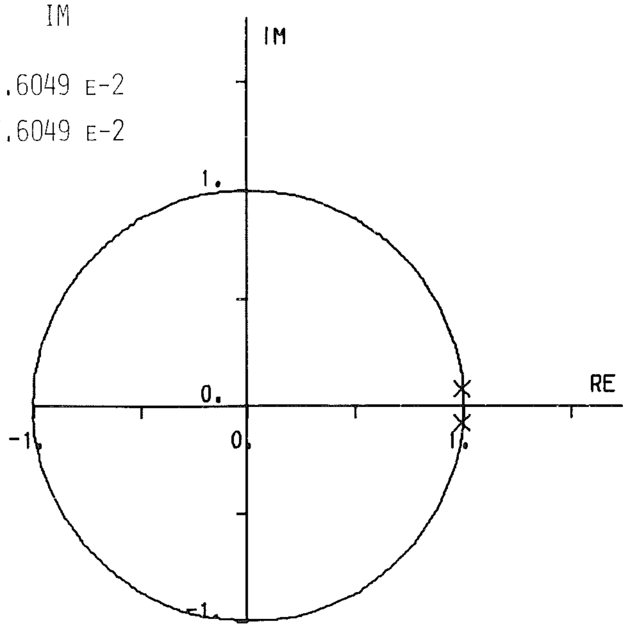


Figure 7.16. The pole configuration for the motor model. The model is stable but poorly damped.

```
1  >SYST BT(BTCPOL) < (MTF) AB Ø.
2  >SYST MT(MTCPOL) < (MTF) AB Ø.
3  >SYST BT(BT(CSS) < (SS) ABC Ø.
4  >SYST MT(MTCSS) < (SS) ABC Ø.
5  >POLY A
6    >INS -1 < Ø.
7    >INS < Ø.
8    >INS < 1.
9    >X
10 >POLY B
11   >INS -1 < 6.75
12   >INS < Ø
13   >X
14 >AGR BTCPOL
15   >INS A
16   >INS B
17   >X
18 >POLY A
19   >INS -1 < Ø.
20   >INS < 1.Ø5
21   >INS < 1.
22   >X
23 >POLY B
24   >INS -1 < 1.895
25   >INS < Ø.
26   >X
27 >AGR MTCPOL
28   >INS A
29   >INS B
30   >X
31 >TRFSS1 (BTCSS) < BT(BTCPOL)
32 >TRFSS1 (MTCSS) < MT(MTCPOL)
33 >SPSS BTBODE < BT(BOCSS) 1 1
34 >SPSS MTBODE < MT(MOCSS) 1 1
35 >BODE (AO) BTBODE BBODE
36 >BODE (AO) MTBODE MBODE
37
38 - - - - - - - - - - - - - - - -
39
40 >ASPEC FISP < FI 3ØØ
41 >ASPEC USP < NU 2ØØ
42 >BODE FISP USP
```

Figure 7.17. Modpac commands used in the second part of the analysis. (Lines 4Ø-42 are actually from Idpac.)

On lines 5-13 and 18-26 the denominator and numerator polynomials for the theoretic ball and motor models. On lines 14-17 and 27-30 the polynomial files are aggregated for use in the two systems BT and MT (theoretic ball and motor model). A continuous time state space representation is then formed for both ball and motor, lines 31 and 32, and then the frequency response is computed, lines 33 and 34, and plotted, lines 35 and 36. The results are shown in Figures 7.18 and 7.19.

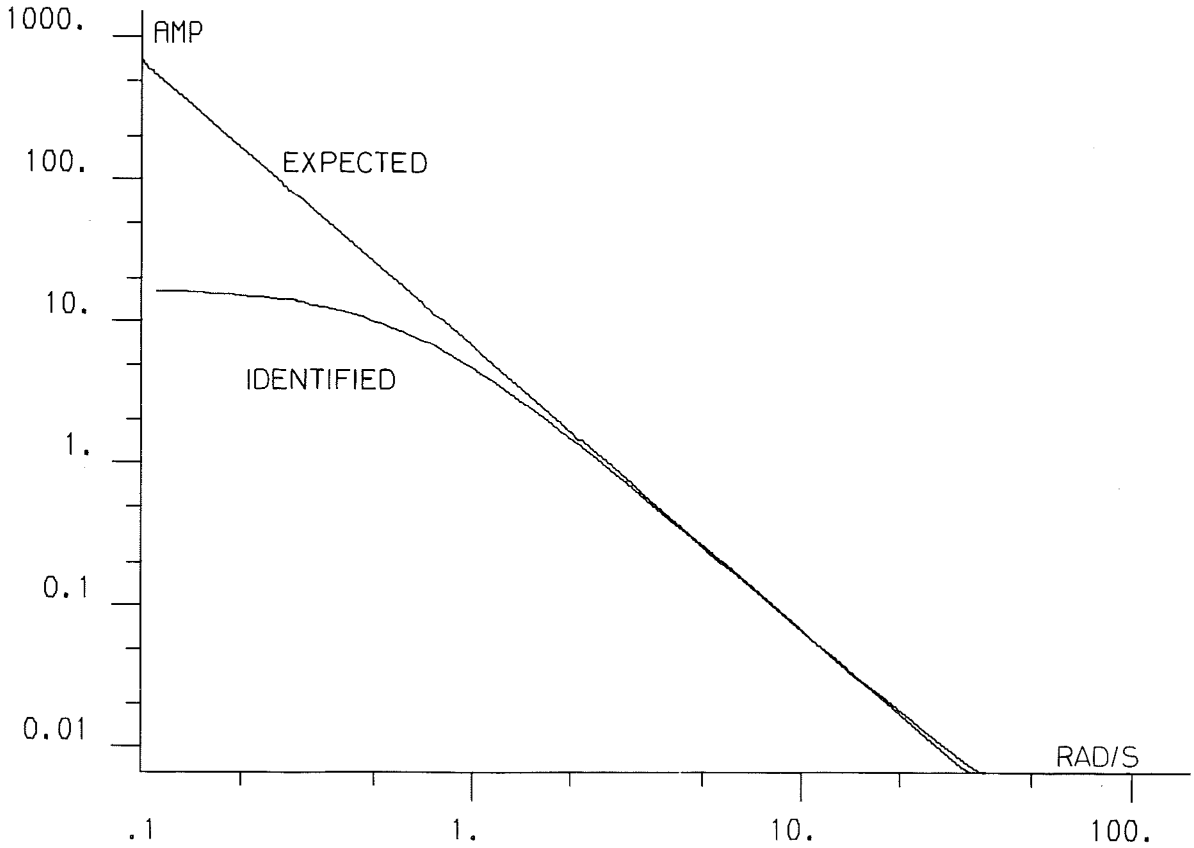


Figure 7.18. The Bode diagram for the identified ball model together with the theoretically expected curve.

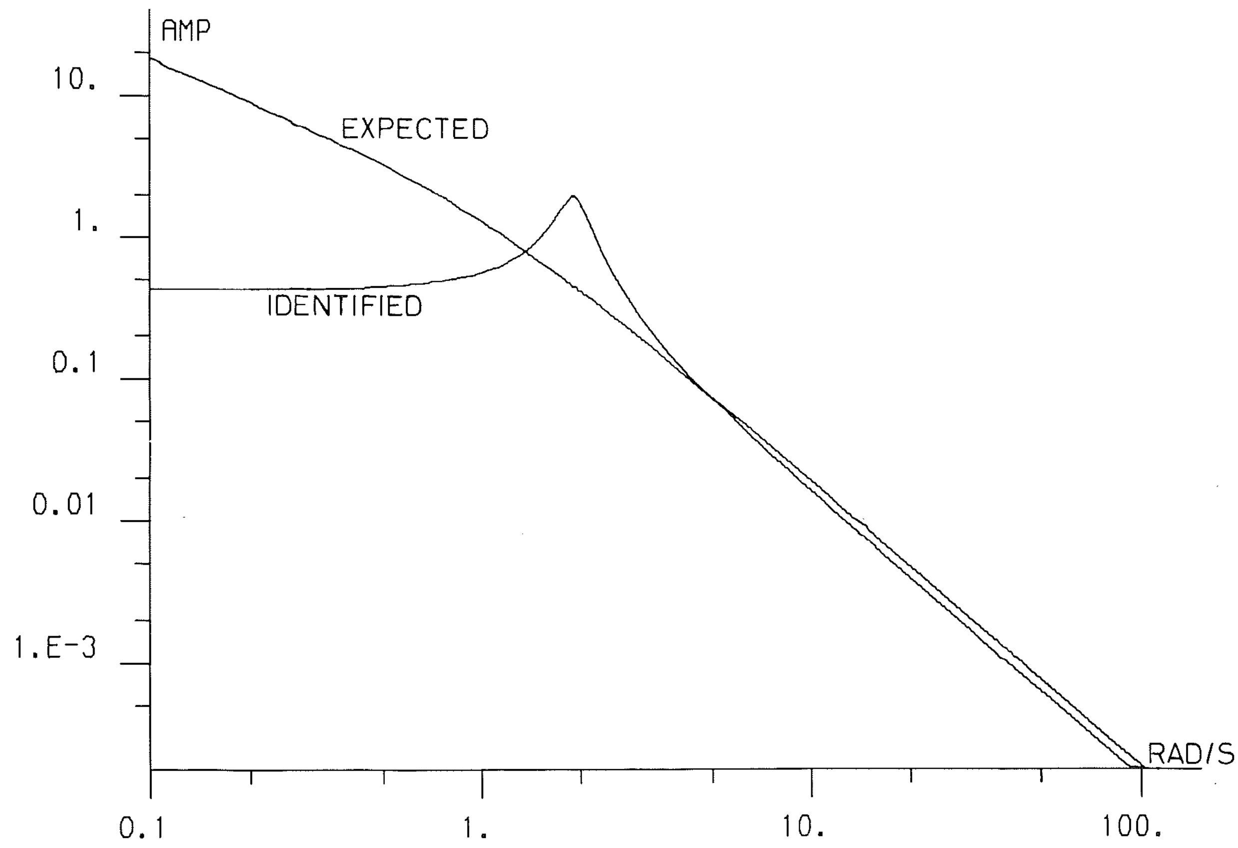


Figure 7.19. The Bode diagram for the identified motor model together with the theoretically expected curve.

The striking characteristic in both cases is the poor agreement between theoretic and identified models for low frequencies, while the high frequency agreement is good to excellent. In order explanain this, the last three commands were executed in Idpac, lines 40-42. The power spectra of the input signals were computed and plotted, Figure 7.20. It is obvious that the identification algorithm succeeded in picking up the correct system dynamics only for the frequency range in which the input had any power. This is also intuitively very natural.

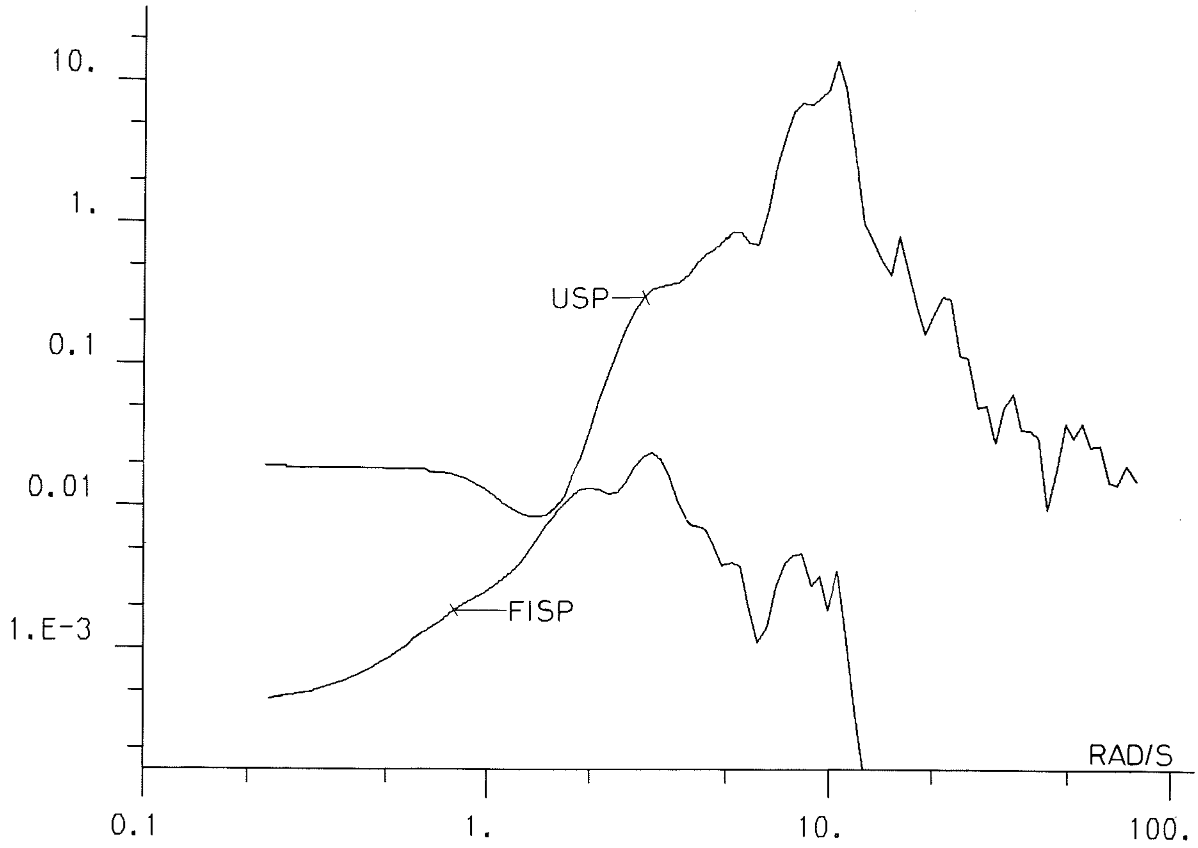
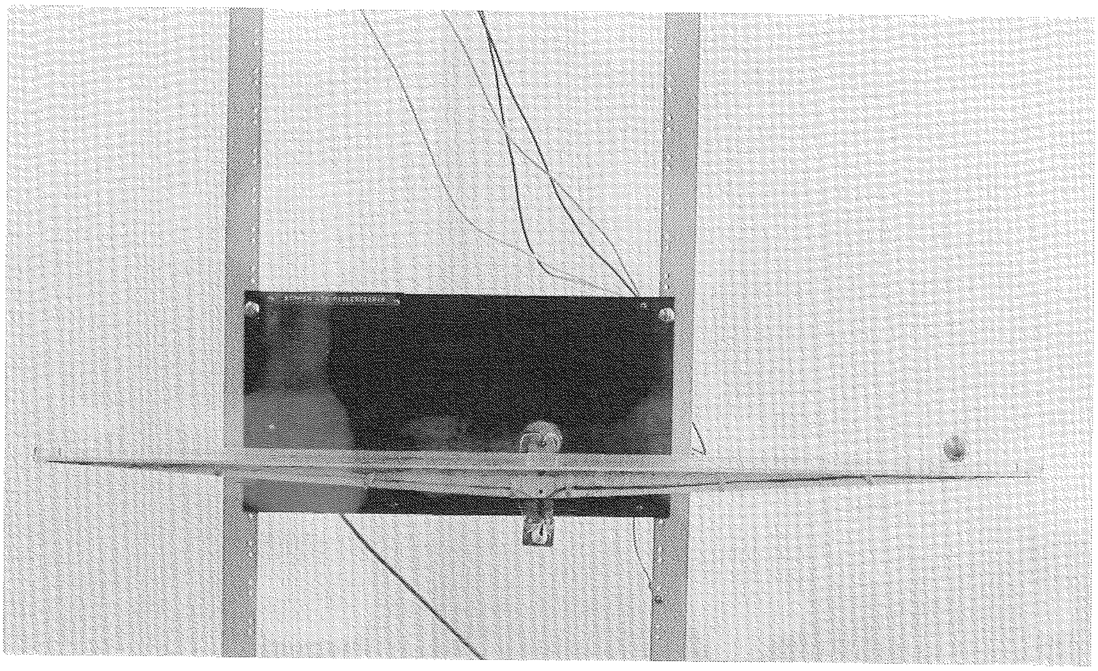


Figure 7.2Ø. The power spectrum for the ball input, i.e. the angle (FISP), and the motor input (USP). Note the low spectral density for low frequencies.

Conclusions

The main objective of this example was, to give a flavour of the use of Idpac and Modpac in solving a practical problem. Some minor details have been omitted and the command sequences have been edited in some places to make the solution somewhat easier to follow. Unfortunately, the creative feeling and the suspense in awaiting the result from the computer are qualities not possible to pass on to the reader. Interactive programs should be run to be fully appreciated.

A lesson learned from this example is that results from identification experiments in closed loop should be judged with care. Models obtained in such a way can only be expected to show the system behaviour for those frequencies in the disturbance that the regulators were unable to cancel. Results with better agreement with the expected results were obtained in other experiments not shown here when the sample interval of the regulators were increased and with the regulators badly tuned.



The ball and beam in operation.
Photo: Rolf Braun.

8. EXAMPLE 2. DESIGN OF A MULTIVARIABLE REGULATOR FOR A CHEMICAL REACTOR

This example will show how Synpac can be applied to the design regulators. The system to be controlled is taken from the literature, [Rose74] and [Munr72]. The design method used there is the INA-method resulting in the use of two simple PI-controllers. As emphasized in Rosenbrock's book, the existing instrumentation of the process, a chemical reactor, called for a design which could be implemented using standard three term pneumatic controllers. We will here endeavour to show that the same goal may be achieved through state space methods. Indeed, the initial system description is given in state space form in the references, so this is no unnatural approach.

One problem is that we have no knowledge available of the relevant physical constraints. Unfortunately, the equations given in the references have been transformed and time scaled. They are given as:

$$\dot{x} = Ax + Bu$$
$$y = Cx$$

with

$$A = \begin{array}{llll} 1.3800 & -0.20770 & 6.7150 & -5.6760 \\ -0.58140 & -4.2900 & 0.00000 & 0.67500 \\ 1.0670 & 4.2730 & -6.6540 & 5.8930 \\ 4.80000E-02 & 4.2730 & 1.3430 & -2.1040 \end{array}$$

$$B = \begin{array}{ll} 0.00000 & 0.00000 \\ 5.6790 & 0.00000 \\ 1.1360 & -3.1460 \\ 1.1360 & 0.00000 \end{array}$$

$$C = \begin{array}{ll} 1.0000 & 0.00000 \\ 0.00000 & 1.0000 \end{array}$$

The A-matrix has the following approximate eigenvalues: +2, 0, -5 and -9. This gives us a hint as to reasonable time constants for the closed loop system. There may very well be modes in the system corresponding to poles to the left of -9, although not included in the model. In order not to interfere with such hypothetical modes, we propose that the control poles of the closed loop system should not have real parts less than -2, say.

Furthermore, we may propose not to allow the control signals to be large compared to the state variables. This will impose the same kind of restriction, namely on the allowable speed of response.

Preliminaries

The program package Synpac, mentioned in Section 6.2, did not exist in its proposed mature form at the time when this example was prepared. This means that some details shown here are not in accordance with conventions discussed earlier, mainly in Sections 5.11 and 5.12. Systems are a mere list of filenames and are restricted to state space representations with A, B, C, and D matrices only. A lossfunction (used in linear quadratic design) is not included in the system representation, but is a separate entity. Aggregates are not implemented. The result is that some things could have been done more elegantly today, but on the other hand, we get an opportunity to discuss some details.

In Figure 5.12, an "extended lossfunction" is defined. The reason is the following: The integrand in the usual quadratic lossfunction looks like

$$x^T Q_1 x + 2x^T Q_{12} u + u^T Q_2 u$$

In simple cases when the state variables have physical significance, the successive change in the matrix elements to obtain a desired behaviour is intuitively natural and straightforward. Very often, however, Q_{12} is not used and Q_1 and Q_2 are kept diagonal. To increase the easily available freedom offered by the method, the extended lossfunction with the following integrand was introduced:

$$x^T QE_1 x + u^T QE_2 u + \dot{x}^T QE_3 \dot{x} + y^T QE_4 y + \dot{y}^T QE_5 \dot{y}$$

Here the outputs, as well as derivatives of state variables and outputs, are directly included and easy to specify in the design. Naturally, the extended lossfunction does not offer anything new, it can (and must) be converted into the standard formulation, but it provides more intuition and ease of use.

In the version of Synpac to come, the extended lossfunction is converted into the standard formulation by a single command, but in the version used here, this facility was not available. By the use of a macro, however, this was easily overcome, see Figure 8.1. The macro PENAL computes Q_1, Q_{12} and Q_2 from the matrices QE_1, QE_2, QE_3 and QE_4, and the system matrices AE, BE and CE. This is an example of a macro used to implement a facility not originially planned by the program designer, cf. Section 3.1. (The present version of MATOP accepts more complex expressions than those used here.)

A few other macros were written when this design example was planned. They will be discussed when used.

Finally, one reason to allow the system representation shown in Figure 5.12 will be apparent in this example. This representation allows a distinction to be made between control inputs, known disturbances, and unknown (stochastic) disturbances, as well as between controlled variables (z)

```
MACRO PENAL
MATOP T<TR AE
MATOP T1<T * QE3
MATOP T<T1 * AE
MATOP Q1<QE1 + T
MATOP Q12<T1 * BE
MATOP T<TR BE
MATOP T<T * QE3
MATOP Q2<T * BE
MATOP Q2<Q2 + QE2
MATOP T<TR CE
MATOP T<TR CE
MATOP T<T * QE4
MATOP T<T * CE
MATOP Q1<Q1 + T
END
```

Figure 8.1. This macro is used in the absense of a corresponding application command to convert an extended loss function into the standard form.

and measured variables (y). Here we will, in the absense of this facility, have to include a new block into the B-matrix in order to allow the input of the impulses used to study the closed loop behaviour. (Line 16 & 17 in Figure 8.2). The design will then be performed in two phases. First a state feedback will be designed giving nice performance for impulse disturbances. In the second phase, this state feedback will be used as a basis for an output feedback from the outputs, their integrals and their derivatives, i.e. a PID regulator structure.

The LQ design phase

The commands used to perform a linear quadratic design for the system in question will be shown here. We start in Figure 8.2 with the initial steps. On line 1 we call the macro START, shown in Figure 8.3. START defines the systems we are going to use and defines the plotting length to 30 samples and the sample interval, used in the simulations, to 0.05 s. Then we move the A, B, and C matrices from backup storage. (DT = magnetic tape).

```
1     >START
      >MOVE A <(DT) A
      >MOVE B <(DT) B
      >MOVE C <(DT) C
5     >MATOP CA < C*A
      >SYST INTEG < AI BI CI NULL XØI
      >ZEROM AI 2
      >UNITM BI 2
      >UNITM CI 2
1Ø    >ZEROM XØI 2 1
      >SYSOP ESYST < CSYST INTEG CA
        >U1 < UR
        >U2 < -Y1
        >U3 < -X1
15      >Y < -Y1 / Y2 / Y3
      >UNITM I6 6
      >EXPAN BT < B (1 1) I6 (1 3)
      >INSI UR 18Ø
        >PULSE
2Ø      >LET IFP. = IFP. + NPLX.
        >PULSE
        >LET IFP. = IFP. + NPLX.
        >PULSE
        >LET IFP. = IFP. + NPLX.
25      >PULSE
        >LET IFP. = IFP. + NPLX.
        >PULSE
        >LET IFP. = IFP. + NPLX.
        >PULSE
3Ø      >X
      >
```

<u>Figure 8.2.</u> The start-up and construction of system representation to work with and the generation of a test input.

In the design, we are interested in the output errors, their integrals and derivatives. In the feedback design we assume the reference signal to be zero so e = -y and $\dot{e} = -\dot{y}$ = -C*A*x. On line 5, CA = C*A is computed and on lines 6-1Ø, a system of two parallel integrators is constructed. On line 11-15, we then build an extended system ESYST from the original one, CSYST, the integrators INTEG and the differentiators CA, to achieve the desired 6 output signals. For use later on, we define the input matrix in TSYST as [B I] and a vector of input signals consisting of unit pulses at suitable intervals.

```
MACRO START
TURN LPCOM ON
SYST CSYST<A B C NULL XØC
SYST CCSYS<AC BC CC DC XØCC
SYST DCSYS<FIC GAMC THC DDC XØDC
SYST CLOSS<NULL Q1 Q12 Q2
SYST ESYST<AE BE CE NULL XØE
SYST LQSYS<ALQ BE CE NULL XØE
SYST TSYST<AE BT CE NULL XØE
SYST PIDSY<APID BPID CPID DPID XØPID
LET NPLX.=3Ø
LET DELTA.=Ø.Ø5
END
```

Figure 8.3. The macro START used to define systems and to initialize.

We are then ready to design a state feedback with a quadratic criterion. The procedure is shown in Figure 8.4. On lines 1-8 all extended lossfunction matrices are zeroed except QE_2, which is set as the unit matrix and QE_4, which is set to punish the two output errors. Then the macro TOTAL, which does the actual job, is called. It is shown in Figure 8.5.

TOTAL first calls PENAL, Figure 8.1, to convert the lossfunction into standard form. Then ITER is called. In ITER we start with a trick. We multiply the lossfunction integrand with $e^{-2*ALPHA*t}$. Thus we force the closed loop eigenvalues to have real parts less than ALPHA, see [Ande71], p. 5Ø. (Again, the present version of MATOP would have accepted a more concise formulation.) Thus the matrix ALQ is computed:

ALQ = A - ALPHA * I

Then we solve the LQ problem for the system LQSYS, where ALQ substitutes AE in the system ESYST.

```
>ZEROM QE1 6
>UNITM QE2 2
>ZEROM QE3 6
>ZEROM QE4 6
>ALTER QE4
  >1 1  1.
  >2 2  1.
  >X
>TOTAL -1.
>MOVE YI1 < YI
>ALTER QE4
  >1 2  -1.
  >2 1  -1.
  >X
>TOTAL -1.
>MOVE YI2 < YI
>ALTER QE4
  >5 5  1.
  >6 6  1.
  >X
>TOTAL -1.
>ALTER QE4
  >3 3  1.
  >4 4  1.
  >3 4  -1.
  >4 3  -1.
  >X
>TOTAL -1.
>MOVE YIF < YI
>PLOT YI1(1 2) YI2(1 2) YIF(1 2)
```

Figure 8.4. The design of a state feedback using linear quadratic theory. The macro TOTAL is shown in Figure 8.5.

The macro TOTAL then computes the state feedback matrix L, constructs the closed loop system CCSYS with 6 reference inputs and with the two control signals appended to the output vector. The eigenvalues of the closed loop system are computed and printed. Finally the closed loop system is transformed to discrete time form and simulated. The result of the first call to TOTAL was a state feedback L (reproduced with two digits):

$$L = \begin{matrix} 0.97 & 1.3 & 0.55 & -0.61 & -1.29 & -6.1 \\ -4.6 & -0.17 & -2.8 & 2.1 & 3.8 & 0.99 \end{matrix}$$

```
MACRO TOTAL ALPHA
PENAL
ITER ALPHA
END


MACRO ITER ALPHA
MOVE YIOLD<YI
ZEROM TMP 6
ALTER TMP
1 1 ALPHA
2 2 ALPHA
3 3 ALPHA
4 4 ALPHA
5 5 ALPHA
6 6 ALPHA
X
MATOP ALQ<AE - TMP
OPTFB L<LQSYS CLOSS
PRINT L
SYSOP CCSYS<TSYST L
U1(1 2)<-Y2
U1(3 4 5 6 7 8)<UR
U2<X1
Y<Y1 / -Y2
POLES EVAL EVEC<CCSYS
PRINT EVAL
SAMP DCSYS<CCSYS
UNITM GAMC 6 DELTA.
SIMU YI<DCSYS UR
PLOT YI(1 2) YIOLD(1 2) -1.5 Ø.5
TYPE L
END
```

Figure 8.5. The macros TOTAL and ITER.

Figures 8.6 and 8.7 show the transient behaviour of the two error signals following a step change in states 1 and 5 respectively. It was found that e_1 and e_2 showed a tendency to have opposite sign. Therefore their difference was included in the lossfunction, lines 11-14 in Figure 8.4. The resulting curves are shown as '2' in Figures 8.6 and 8.7. Next, to improve the damping the two derivatives were punished, lines 17-2Ø, and finally the difference in integrated error, without much effect, lines 22-27. This

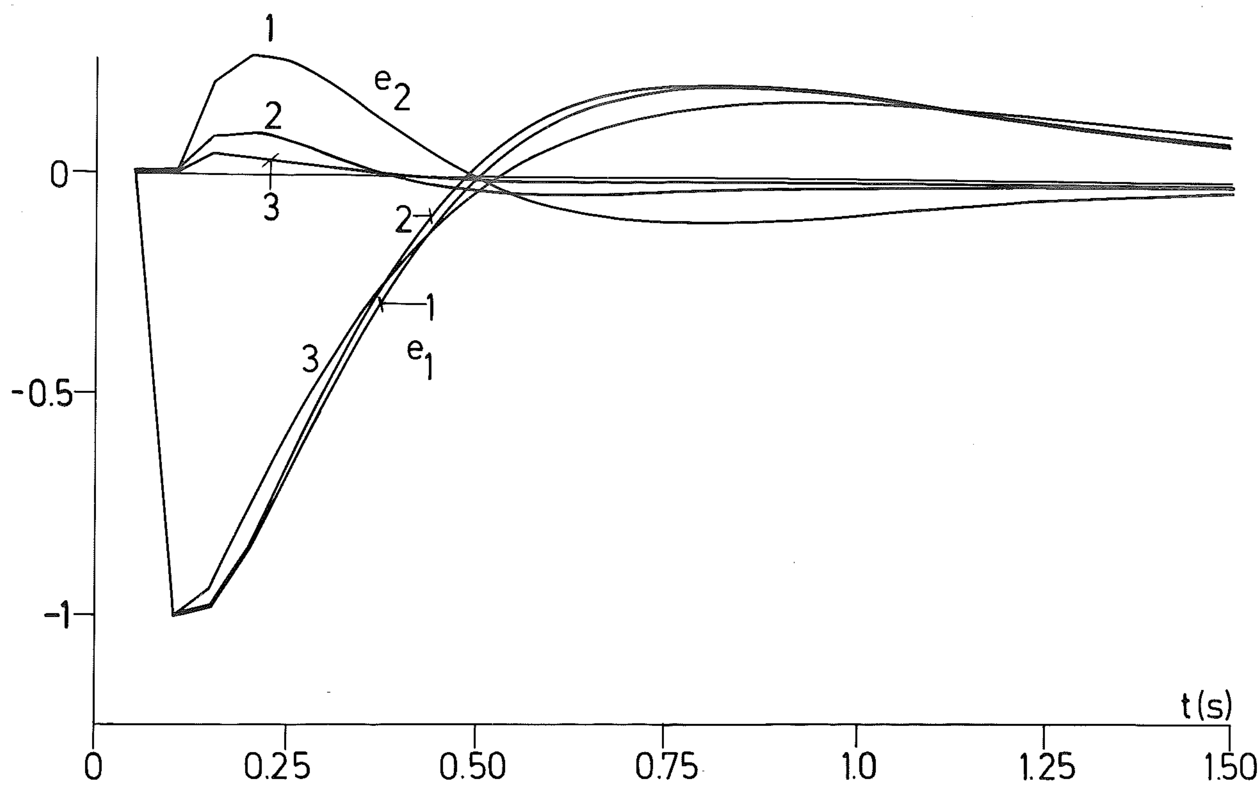


Figure 8.6. The response of the errors $e_i = -y_i$ to a step disturbance in the first state variable for different designs, first (1), intermediate (2) and final (3).

final choice of lossfunction produced the following state feedback L (again with only two digits shown):

$$L = \begin{matrix} 0.30 & 4.3 & 0.22 & -0.53 & 0.00 & -14. \\ -8.0 & 0.19 & -7.0 & 5.5 & 6.6 & -1.8 \end{matrix}$$

We observe immediately that the first control signal will depend mainly on the second state variable and its integral (x_6). Similarly the second control will depend on the first state variable and its related integral (x_5) but also on states 3 and 4, which were not available as outputs. The final curves are marked '3' in Figures 8.6 and 8.7. Note that the interaction was vastly decreased in the first step and that a clearly noticable improvement in damping was achieved in the second step.

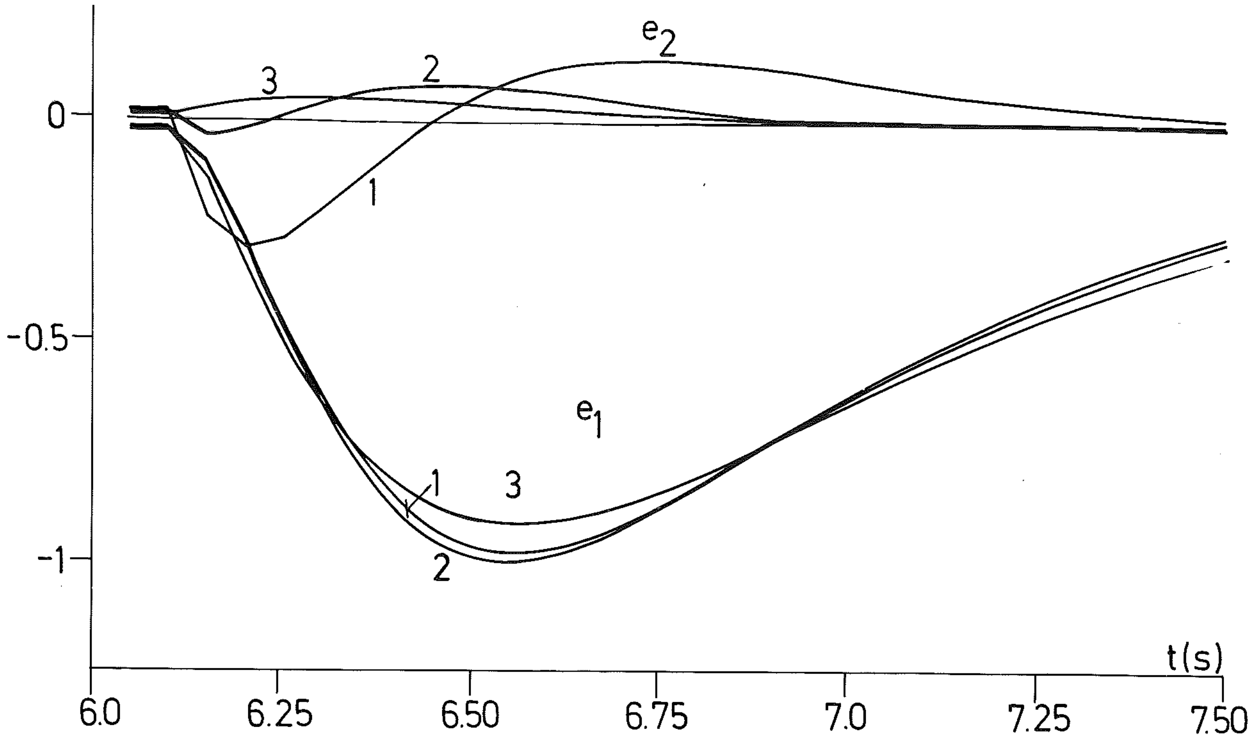


Figure 8.7. The same as Figure 8.6 but with the step disturbance in the fifth state variable (i.e. the integral of e_1).

Of course, some other choices in design parameters were explored in the preparation of this example. Similarly, the behaviour of the control signals as well as the responses to other disturbancies were studied, although not shown here.

The PID design phase

The specifications called for a simple control scheme which should be implemented with pneumatic three term controllers. We will therefore try to convert the state feedback obtained into an output feedback with PID regulators. This is done through the method given in [Beng73]. The macro PIDGE, shown in Figure 8.8, exploits the fact that the outputs from the system was chosen as the error, error integral, and error derivative. Thus the conversion of the state feedback into an output feedback effectively gives the parameters in a set of PID-regulators. The details of PIDGE are then: first the resulting response of a previous design is saved, then the output feedback is computed for a specified feedback structure (i.e. choice of available outputs). The resulting matrix, here called LR is printed. Then the closed loop system is formed as before. Note that the derivative of the reference input is not included in the forming of the error derivative. Finally the closed loop eigenvalues and the closed loop system matrices are printed and the system is simulated and the response is plotted.

Figure 8.9 then shows the commands entered to perform the design. First of all we increase the time scale on the plots (line 1) and define two step inputs (actually long pulses so as to allow the system to return to the zero state). Then we define the weighting matrix for the reduced feedback command together with a matrix specifying the feedback structure.

```
MACRO PIDGE
MOVE DK YPIDO<DK YPID
REDFB LR<ESYST L FBS W
TYPE LR
PRINT LR
SYSOP PIDSY<CSYST INTEG CA LR
U1<-Y4
U2<UR-Y1
U3<-X1
U4(1 2)<UR-Y1
U4(3 4)<Y2
U4(5 6)<Y3
Y <Y1(1 2) / -Y4
POLES PIDEV EVEC<PIDSY
PRINT PIDEV
PRINT APID
PRINT BPID
PRINT CPID
PRINT DPID
SAMP DCSYS<PIDSY
SIMU YPID<DCSYS URPID
PLOT YPID(1 2) URPID
END
```

Figure 8.8. The macro PIDGE, generating a PID-type feedback from a given state feedback.

Initially we have equal weighting and include all possible outputs. We then call PIDGE, line 12. The resulting output feedback matrix is (two significant digits):

$$LR = \begin{pmatrix} 0.049 & -2.1 & 0.0043 & -14. & -0.032 & 0.52 \\ 6.9 & 3.0 & 6.6 & -1.8 & 1.0 & 0.69 \end{pmatrix}$$

the closed loop eigenvalues remain unchanged. The most striking property is though that output 1 of the original system does not seem to influence the first control signal very much (recall the order of the extended system outputs: output errors, their integrals, and their derivatives). The

```
1   >LET NPLX.=60
2   >INSI URPID 240
3     >LET IFP.=1
4     >PULSE 60
5     >LET IFP.=121
6     >PULSE 60
7     >X
8   >UNITM W 6
9   >FBS 2 6
10    #1 1 1 1 1 1
11    #1 1 1 1 1 1
12  >PIDGE
13  >ALTER FBS
14    >1 1 0
15    >1 3 0
16    >1 5 0
17    >X
18  >PIDGE
19  >ALTER FBS
20    >2 2 0
21    >2 4 0
22    >2 6 0
23    >X
24  >PIDGE
25  >ALTER FBS (1 6) 0
26  >ALTER W
27    >1 1 10.
28    >2 2 10.
29    >X
30  >PIDGE
```

Figure 8.9. The commands used to reach the desired control configuration.

obvious next step is therefore to exclude these coefficients completely, lines 13-17. The next call to PIDGE results in a new value for LR:

$$LR = \begin{matrix} 0.0 & -3.6 & 0.0 & -11. & 0.0 & 0.12 \\ 6.9 & 3.0 & 6.6 & -1.8 & 1.0 & 0.69 \end{matrix}$$

Note that the coefficients for output 2 do not change (actually not within 5 digits). The eigenvalue closest to the origin moved from -1.75 to -1.41.

Next we try to exclude the second output from the second control signal, lines 19-24. The resulting feedback has a comparatively small contribution from the second error derivative, so we try to eliminate it (effectively introducing a PI regulator rather than a PID). In the same time, we increase the weight put on the two eigenvalues closest to the origin, lines 26-19. The final call to PIDGE (line 30) produces the following LR:

$$LR = \begin{matrix} 0.0 & -4.0 & 0.0 & -9.9 & 0.0 & 0.0 \\ 6.7 & 0.0 & 8.2 & 0.0 & 1.0 & 0.0 \end{matrix}$$

The eigenvalue closest to the origin has now moved to -1.25.

Some of the step responses obtained for the initial and final design are shown in Figures 8.10 and 8.11. The control signals for the final design are shown in Figures 8.12 and 8.13. The step responses for the state feedback are included.They were computed for the closed loop system obtained through:

```
8: 200


>SYSOP CCSYS<CSYST INTEG L
  >U1<-Y3
  >U2<Y1-UR
  >U3(1 2)<Y1-UR
  >U3(3 4)<X1(3 4)
  >U3(5 6)<X2
  >Y<Y1 / -Y3
>
```

An Observation

It can be observed in the responses that least interaction was achieved for the state feedback regulator. The theory behind the reduced feedback command says that modes corresponding to unchanged eigenvalues also remain unchanged. Thus one might expect the step responses for the state feedback regulator and the first PID regulator structure to be identical. That this is not the case is due to the fact that the reference input enters differently into the closed loop systems. The result is that the zeroes of the system change. Similarly, the fact that the step responses improve when some feedback loops are cut out is a fortunate effect and depends on the properties of the original system.

Finally, it may be noted that it would have been quite possible to use a dynamical system more closely approximating the D part of a real life controller, instead of the static system used here.

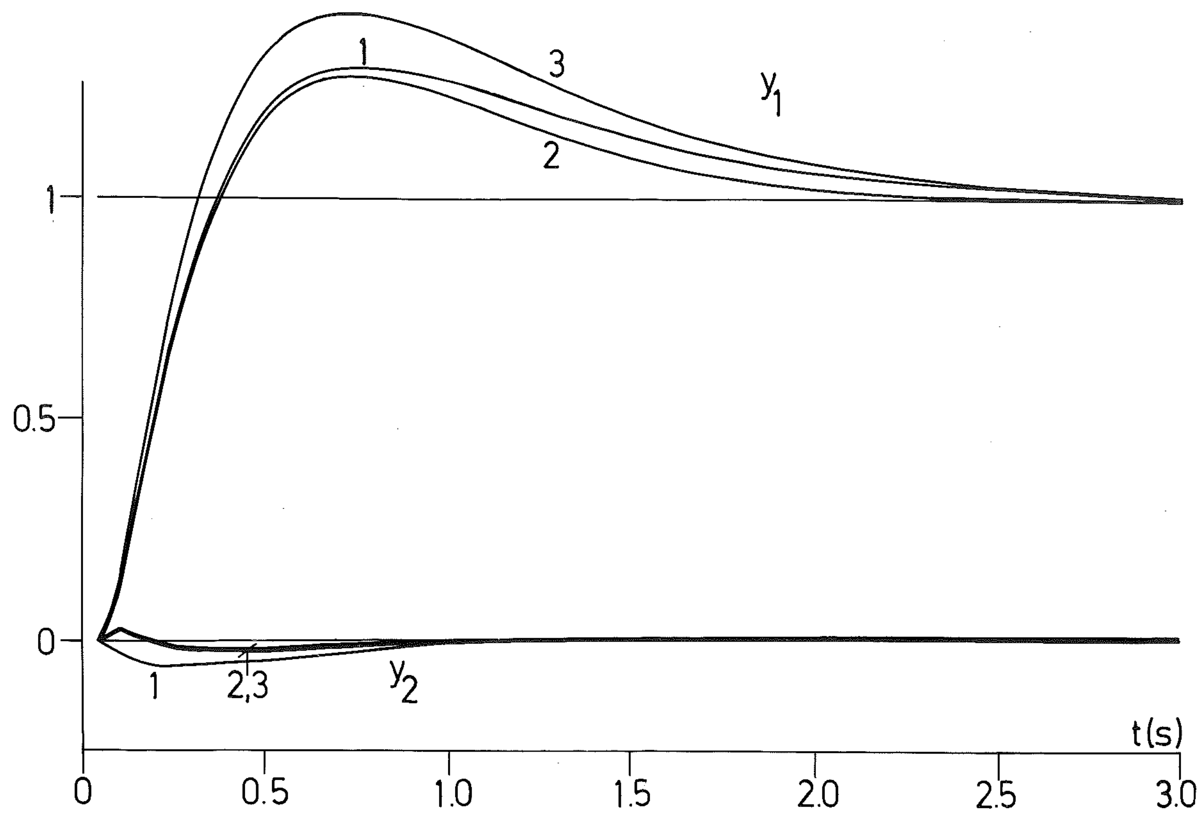


Figure 8.1Ø. The responses for a step command in y_1. (1) is the first PID-design, (2) the final one. (3) is the state feedback.

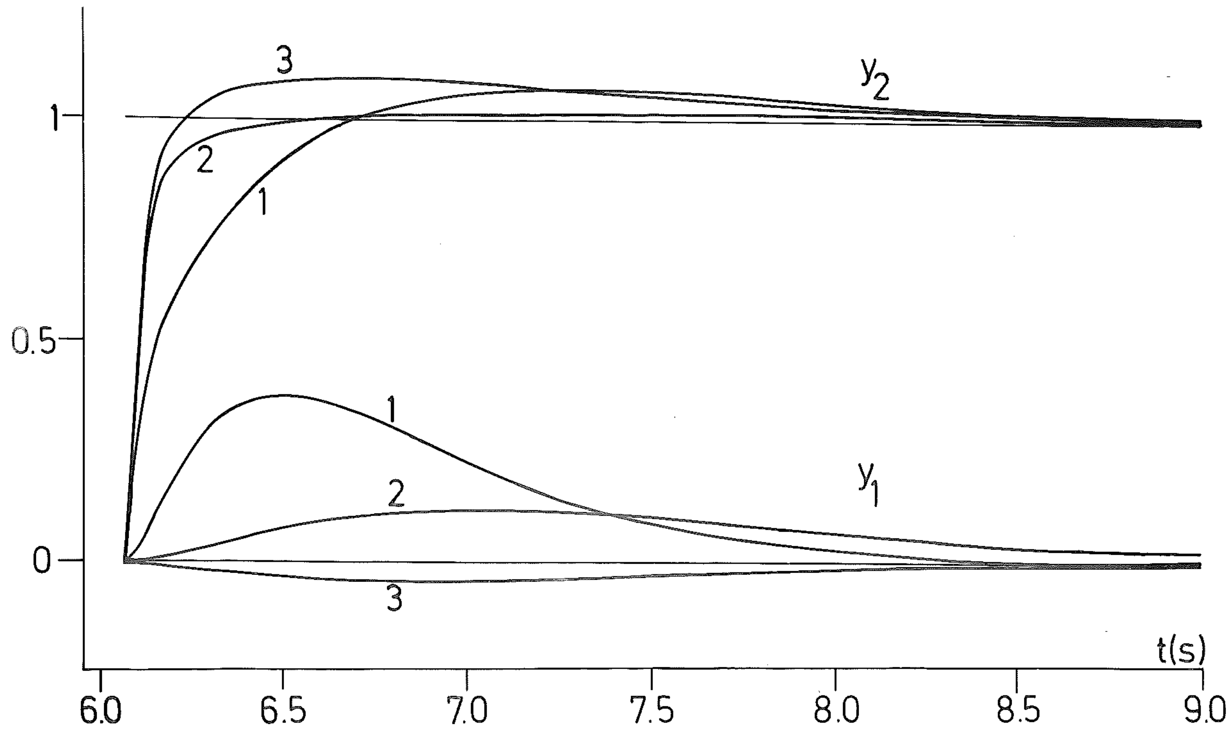


Figure 8.11. Otherwise similar to 8.1Ø, the step is now in y_2.

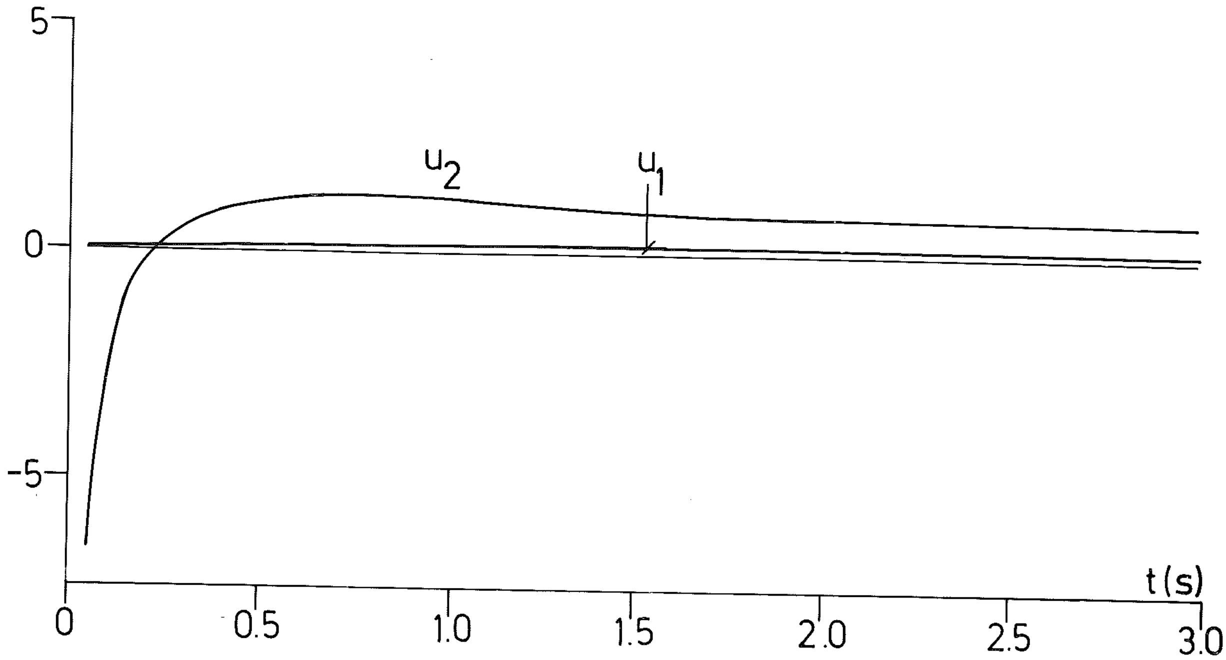


Figure 8.12. The two controls u_1 and u_2 for the final PID-design when there is a step demand in y_1.

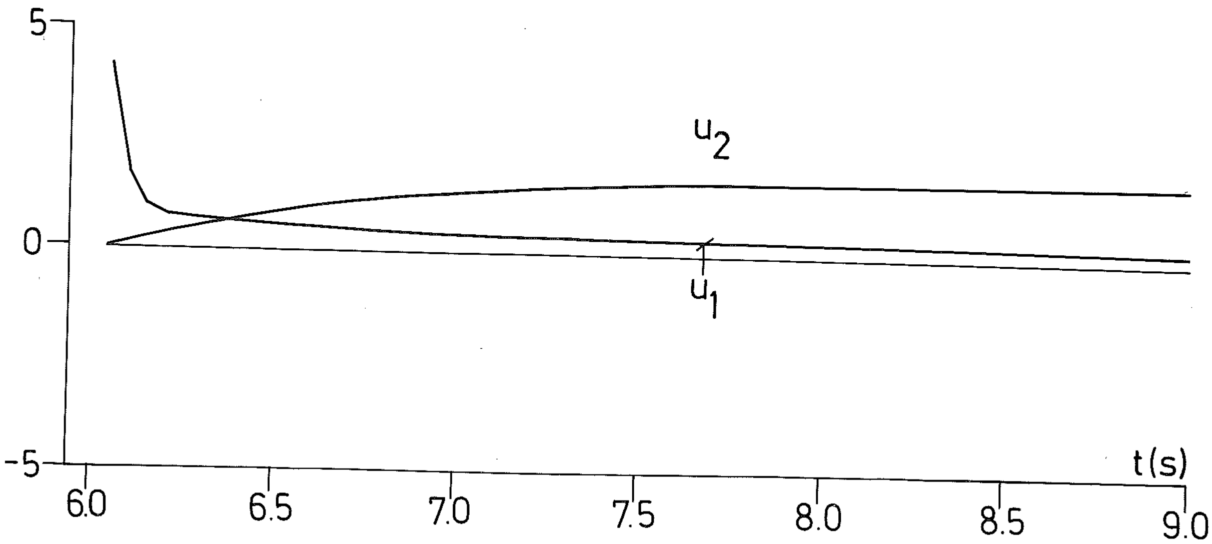


Figure 8.13. The continuation of 8.12 with the step in y_2.

Conclusion

The purpose of this example was to give some flavour of the power attainable with an interactive design program. In this case, much of the usefulness of the program did not lie only in the availability of the numerical algorithms as such, but to a large extent in the way relevant data structures could be used and constructed. Specifically, the possibility to be able to connect a given system with suitable subsystems to form various closed loop systems, was of main importance.

Another factor of great importance for this kind of work is the macro facility. Here, a macro was used to initialize the program with some problem dependent information. The main use of macros was, however, to allow the iterative loops in the design scheme to be defined in a form, simple to use, simple to modify, and adapted to the specific problem at hand.

Finally, a lesson learned from this example is that it is indeed possible to construct regulators simple to implement using linear quadratic design. The key to this possibility was the derivation of an output feedback according to the method given in [Beng73].

9. REFERENCES

[Aaro77] 47, 48
I. Aaro: Intraction Models. TRITA-NA-7704. Department of Information Processing and Computer Science. The Royal Institute of Technology, Stockholm, Sweden.

[Ande71] 192
B.D.O. Anderson, J.B. Moore: Linear Optimal Control. Prentice-Hall.

[Asimov] 71
I. Asimov: Foundation. 1951.

[Beng73] 193, 203
G. Bengtsson: A Theory for Control of Linear Multivariable Systems. LUTFD2/(TFRT-1006). Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[Birt73] 116
G.M. Birtwistle, O.J. Dahl, B. Myrhaug, K. Nygaard: SIMULA BEGIN. Studentlitteratur - Auerbach.

[Bosl72] 24
M.J. Bosley, F.P. Lees: A Survey of Single Transfer Function Derivations from High Order State Variable Models. Automatica, Vol. 8, Nr. 6.

[Doyl79] 37
J.G. Doyle: Robustness of Multiloop Linear Feedback Systems. 17:th IEEE Conference on Decision and Control. San Diego.

[Elmq72] 148
H. Elmqvist: SIMNON - Ett interaktivt Simuleringsprogram for Olinjara System. (Master Thesis). LUTFD2/(TFRT-5113). Department of Automatic Control, Lund Institute of Technology, Lund, Sweden. (In Swedish)

[Elmq75] 27, 40, 46, 131, 148, 156
H. Elmqvist: SIMNON - An Interactive Simulation Program for Non-linear Systems. LUTFD2/(TFRT-3091). Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[Elmq et al 76] 155
H. Elmqvist, A. Tyssø, J. Wieslander: Programming and Documentation Rules for Subroutine Libraries. The Scandinavian Council for Applied Research.

[Elmq78] 27, 40
H. Elmqvist: A Structured Model Language for Large Continuous Systems. LUTFD2/(TFRT-1015). Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[Esse77a] 153
T. Essebo: Character and String Handling in Intrac. Programming manual. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[Esse77b] 157
T. Essebo: File Handling in Program Packages. Programming manual. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[Gust73] 151
I. Gustavsson, S. Selander, J. Wieslander: IDPAC - User's Guide. LUTFD2/(TFRT-3Ø61). Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[Gust et al 74] 169
I. Gustavsson, L. Ljung, T. Söderström: Identification of Linear, Multivariable Process Dynamics Using Closed Loop Experiments. LUTFD2/(TFRT-3Ø69). Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[Hall78] 2Ø, 46
L-G. Hallström: Generella Beräkningar pa Mätdata. (Master Thesis). LUTFD2/(TFRT-521Ø). Department of Automatic Control, Lund Institute of Technology, Lund, Sweden. (In Swedish)

[Iver62] 161
K.E. Iverson: A Programming Language. Wiley.

[Jens74] 116
K. Jensen, N. Wirth: Pascal - User Manual and Report. Springer-Verlag.

[Jens76] 2Ø
L. Jensen: DATPAC - Programpaket for Mätdataanalys. BKL 1976:13. Department of Building Function Theory, Lund Institute of Technology, Lund, Sweden. (In Swedish)

[Jons71] 147
B. Jönsson: Konservativt Datorprogram för Processidentifiering. (Master Thesis) LUTFD2/(TFRT-5Ø91). Department of Automatic Control, Lund Institute of Technology, Lund, Sweden. (In Swedish)

[MFar73] 37
A.G.J. MacFarlane, J.J. Belletrutti: The Characteristic Locus Design Method. Automatica, Vol. 9, p 575.

[MFar77] 35 A.G.J. MacFarlane, I. Postlethwaite: The Generalized Nyquist Stability Criterion and Multivariable Root Loci. International Journal of Control.

[Munr72] 187
N. Munro: Design of Controllers for Open-loop Unstable Multivariable System Using Inverse Nyquist Array. Proc. IEE. Vol. 119.

[Nove72] 148
T. Noven: SYNPAC - Ett Interaktivt Program för Syntes av Reglersystem. (Master Thesis). LUTFD2/(TFRT-5104). Department of Automatic Control, Lund Institute of Technology, Lund, Sweden. (In Swedish)

[Rose69] 37
H.H. Rosenbrock: Design of Multivariable Control Systems Using the Inverse Nyquist Array. Proc. IEE. Vol 116.

[Rose74] 187
H.H. Rosenbrock: Computer-Aided Control System Design. Academic Press.

[Rose75] 71
H.H. Rosenbrock: The Future of Control. Sixth IFAC Congress. Boston.

[Shan77] 161
S. Shankar, D.P. Atherton, D.G. MacNeil: Computer Aided Design of Control Systems Using APL. IFAC Symposium on Trends in Automatic Control Education, Barcelona.

[Scho77] 158
T. Schönthal: Implementation Procedures, Plot Routines. Programming manual. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[Wies70] 147
J. Wieslander: Matpac - A Conversational Matrix Program. Unpublished material.

[Wies73] 158
J. Wieslander: ABSOLU and RANDEX, Fast Overlays Using Random Access. Programs submitted to DECUS.

[Wies76]
J. Wieslander: IDPAC - User's Guide, Revision 1. LUTFD2/(TFRT-3099). Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[Wies78] 12, 73, 89, 151
J. Wieslander, H. Elmqvist: INTRAC - A Communication Module for Interactive Programs - Language Manual. LUTFD2/(TFRT-7132). Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

APPENDIX

This appendix contains an abbreviated list of the commands in the programs Idpac, Modpac and Synpac. A few commands of a specialized nature have been omitted, and in some cases, the most general form is not shown. This list is included for two reasons. It will give information on the facilities available in these programs, and secondly, it will serve as a reference in reading the examples in Chapter 7 & 8.

A list of the commands available within Intrac is not given here. Those commands were described in detail in Section 4.6.

The list also shows the command syntax. Optional arguments are contained within [ ], and the argument names should suggest their meaning. Many commands are available in more than one program. Therefore, commands in Modpac also found in Idpac are not repeated in the Modpac list. Similarly, the commands in the Synpac list have a different form or are not found in either of the other two programs.

Idpac Command List

ACOF

Computes the autocovariance for a column in a data file

```
ACOF FNAM1[(C1)] < FNAM2[(C2)] NOL [EXT]
```

ASPEC

Computes the autospectrum for a column in a data file

```
ASPEC FRF[(F)] < FNAM2[(C2)] NOL [FREQ]
```

BODE

Plots amplitude and phase versus angular frequency in a logarithmic diagram on display

```
BODE [(SW)] FRF1[(F11 F12 ..)] [FRF2[(F21 .. )] ..
```

Subcommands:

```
PAGE
KILL
```

CCOF

Computes the cross covariance for a column in a data file

```
CCOF FNAM1[(C1)] < FNAM2(C21 C22) NOL

CCOF FNAM1[(C1)] < FNAM2[(C2)] FNAM3[(C3)] NOL
```

CONC

Concatenates two data files

```
CONC [DNAM1] < DNAM2 DNAM3
```

CONV

Transfers a free format data file to a binary data file

```
CONV DNAME < FNAM[(C1..)] NCOLX [TSAMP]
```

CSPEC

Computes the cross spectrum for a column in a data file

```
CSPEC FRF[(F)] < FNAM2(C21 C22) NOL [IALIGN] [FREQ]

CSPEC FRF[(F)] < FNAM2[(C2)] FNAM3[(C3)] NOL [IALIGN]
                 [FREQ]
```

CUT

Picks out a part of a data file

```
CUT [DNAM1] < DNAM2 IB IE
```

DELET

Deletes files from disk

```
DELET FNAM1[(DMODE1)] [FNAM2[(DMODE2)] ... ]
```

DETER

Performs simulation of a multiple input - single output linear discrete dynamic system as an addition of simulations of single input - single output systems

```
DETER DNAM1[(C1)] < SNAME[(NAME)] DNAM2[(C21 ..)]
                    [DNAM3[(C31 ..)] [....]..] [NP]
```

DFT

Performs the Discrete Fourier Transform on a time series

```
DFT [(RES)] [(WND)] SPEC < DATA[(IND)] [START NSAMP]
```

DSIM

Similar to DETER but includes a noise input

```
DSIM DNAM1[(C1)] < SNAME[(NAME)] DNAM2[(C21 ..)]
                   [DNAM3(C31 ..)] [....]..] [NP]
```

EDIT

Symbolic text editor

```
EDIT FNAME
```

FHEAD

Displays file head and enables the user to change its parameters

```
FHEAD [AGGREG:]FILE
```

FILT

Computes a digital low- or high-pass Butterworth filter of given order and with given cut-off frequency. A band pass/stop filter is constructed by combining a high- and a low-pass filter and has the double order.

```
FILT PNAME < FITYP NO DELTAT OML [OMH]
```

FORMAT

Converts a binary data file into a formatted data file

```
FORMAT [FFILE] < BFILE[(C1 C2 .. )] [BEGIN COUNT]
```

FROP

Adds, subtracts, multiplies or divides two frequency response files for frequencies which coincide with an error less than .000001.

```
FROP [FRF1[(F1)]] < FRF2[(F2)] OP FRF3[(F3)]
```

FTEST

Performs a file existence test

```
FTEST FNAME [(DMODE)]
```

GETFIL

Retrieves a file from back-up storage

```
GETFIL PROGFILE FILESPEC [FILESPEC..]
```

IDFT

Performs the Inverse Discrete Fourier Transform on a frequency response

```
IDFT DATA < SPEC[(IND)]
```

INSI

Generates data sequencies

INSI FNAME [(C)] NP [TSAMP]

Subcommands:
PRBS [IBP [NBIT [ISTART [OPT]]]]
NORM [RMEAN SIGMA]
RECT [A B]
SINE [OMEGA FI]
ZERO
STEP
RAMP [A B]
PULSE [LENGTH]
SRTW [PS]
LOOK
KILL
X

LIST

Lists on display, line printer or teleprinter the contents of (a part of) a data file, a macro file or a system file - for a data file the columns and the first record and number of records may be specified, for a system file sections of interest may be specified

LIST [(DEV)] [(FEED)] [(DMODE)]
[AGGREG:]FNAME[(A1 A2..)] [IF NUM]

LS

Performs Least Squares identification

LS [(SW)] SYST[(SECT)] < SFIL [EXT]

Subcommands:

SAVE STDEV
SAVE COMAT
KILL
X

ML

Performs Maximum Likelihood identification

ML [(SW)] SYST[(NAME)] < DATA[(C1 ..)] NO [EXT]

Subcommands:

INVAL 'ABC'/'C' SYST[(NAME)]
FIX A (2) [VA2] (3) [VA3] B (21) [VB21] .....
SAVE [STDEV] [GRAD] [EVALS] [COMAT]
LOOK
KILL
X

MOVE

Transfers a data file, a system file, a macro file or specified columns in a data file from one kind of mass storage to another. Can also be used to rearrange the columns of a data file

```
MOVE [(OUTP)] [(DMODE)] [[AGOUT:] FOUT [(C11..)]] [O] <
     [(INP)] [AGIN:] FIN [(C21..)]
```

PICK

Picks out equidistant records from a data file

```
PICK FNAM1 < FNAM2 NR
```

PLMAG

Makes it possible to plot small parts of a data vector and alter data values or remove data points

```
PLMAG DATA [(C)]
```

Subcommands:

```
B[LOCK] NB
P[LBEG] NR
A[LTER] NR [NUM]
PA[GE]
D[ELET] NR [NUM]
KILL
X
```

PLOT

Plots data files on display

```
PLOT [(NP)] [FNAMX[(C1..)] < ] [(OPT1)] FNAM1[(C11..)]
    [[(OPT2)] [FNAM2[(C21..)]] .. ] [YMI YMA]
```

Subcommands:

```
KILL
PAGE
SKIP [N]
```

RANPA

Generates a Gaussian random vector with given covariance matrix and adds it to the parameters in a system description

```
RANPA SNAM1 < SNAM2[(NAME)]
```

RESID

Computes residuals, autocorrelations of residuals, and cross correlations between residuals and input signal(s)

```
RESID RES[(C1)] < SYST[(NAME)] DATA[(C11 C12 .. )]
                  [NOL [NFREE]] [EXT]
```

Subcommands:

KILL
PAGE
TABLE

SAVFIL

Saves a file on back-up storage

```
SAVFIL PROGFILE FILESPEC [FILESPEC..]
```

SCLOP

Each element in a specified column in a data file is added, subtracted, multiplied or divided by a constant

```
SCLOP [FNAM1[(C1)] < FNAM2[(C2)] OPER CONST
```

SLIDE

Shifts the columns in a data file along each other

```
SLIDE [FNAM1] < FNAM2 K1 K2 K3 ..
```

SPTRF

Computes the power spectrum or the amplitude and phase of a transfer function TPN(Q*-1)/TPD(Q*-1)

```
SPTRF [(SW)] FRF[(F1)] < SYST[(NAME)]
      TPN[(NRN)] / TPD[(NRD)] [FREQ[(F2)]]
```

SQR

Computes square-root matrix for LS identification

```
SQR RFIL < FNAME [(C1 C2 ..)] SFIL
```

STAT

Computes the statistical properties sum, mean value, variance, standard deviation, minimum and maximum value for a specified column in a data file

```
STAT FNAME [(C)] [EXT]
```

STRUC

Creates and updates struc files

```
STRUC SNAM2

STRUC [SNAM2] < SNAM1
```

Subcommands:

```
REVERT
NA    [SW] NVA1
NU    [SW] NVA1
NB    [SW] NV1 ... NVNU
KB    [SW1] NV1 ... NVNU
FIX A(N) [VN] (M) [VM] ...
B NU1 (N1) [V1] (N2) ... B NU2 ...
UNFIX A(..  N .. M ..) B NU1 (N1 ... NN) ..
SW : 'MAX' / 'ACT'
SW1: SW / 'MIN'
KILL
X
```

TREND

Removes polynomial trends from data vectors using least-squares technique

```
TREND [FNAM1[(C1)]] < FNAM2[(C2)] NO [IF IL]
```

TURN

Manipulates program switches

```
TURN SWITCH STATE
```

VECOP

Adds, subtracts, multiplies or divides two data vectors element by element

```
VECOP [DNAM1[(C1)] < DNAM2[(C2)] OPER DNAM3[(C3)]
```

Modpac Command List

AGR

Creates and updates an aggregate file

AGR AGROUT

AGR [AGROUT] < AGRIN

Sub-commands:

LOOK [NAME]
KILL
X
LOC NAME
INS NAME
REP [NAME]
DEL
ISO
TOP
BOT
REM
ADV [NR]

ALTER

Enables the operator to alter matrix elements

ALTER [AGGREG:] MATRIX [(IR IC) VALUE]

Subcommands:

KILL
X

BODE

Plots amplitude and phase versus angular frequency in a logarithmic diagram on display

BODE [(SW)] FRF1[(F11 F12 ..)] [FRF2[(F21 ..)] ..

Subcommands:

PAGE
KILL

CONT

Computes the system matrices for a continuous version of a discrete linear dynamic system

CONT [SYSOUT][(NAMOUT)] < SYSIN[(NAMIN)] [EPS]

ENTER

Creates a matrix file

ENTER [AG:]MAT NR [NC] [TSAMP]

Subcommands:

KILL
X

EXPAN

Creates a new matrix from any number of old matrices. It may be specified where in the new matrix the upper left corner of the old matrices shall be placed.

```
EXPAN [[AG1:]M1] < [AG2:]M2[(IX2 IY2)]
                   [[AG3:]M3[(..)]..]
```

KALD

Decomposes a given system into subsystems according to controllability and observability. The result of the decomposition may be viewed schematically, and the user may save parts of interest by means of subcommands.

KALD SNAME[(NAME)] [AEPS REPS]

Subcommands:

SAVE RNAME[(NAME)] < RESLT [ATTR1 [ATTR2]]
LOOK
X

LUEN1 / LUEN2

The Luenberger observer commands are structured in three steps, LUEN1, a pole assignment command and LUEN2.

In LUEN1 a system transformation is performed in order to:
1. Check if all measurements are independent , i.e. test the Rank(C)
2. Rearrange the rows of C to get the form C=(Ø I) the transformation matrix T is calculated

In the next stage a matrix K is calculated in order to achieve desired poles of the system A11-K*A21

In LUEN2 T and K are given and the observer matrices are calculated

LUEN1 T SYST1 < SYST2[(NAME2)] [EPS]

LUEN2 SYST1 < SYST2[(NAME2)] T K [EPS]

MATOP

Evaluates matrix expressions

MATOP [(EXT)] [[AGGREG:]MATRIX] < matrix expression

NIC

Plots Nichols curves on display

NIC [WMIN WMAX] FRF1[F11 ..)] [FRF2...]

NYQ

Plots Nyquist curves on display

NYQ [WMIN WMAX] FRF1[(F11 ..)] [FRF2 ...]

PLEV

Plots eigenvalues on display and enables the operator to alter them. The numerical values of the eigenvalues are also displayed if there is room for them on the screen.

PLEV FNAM2 [FNAM3 FNAM4 ....]

PLEV FNAM1 < FNAM2

Subcommands:

ALT N VR [VI]
ALT N1 VR VI _N2
SCALE N V
DAMP N Z
EXAM N
LOOK
X
KILL

POCONV

Converts Miso Transfer Function models from polynomial image form to polynomial file form and vice versa

POCONV [SYSOUT][(NAMOUT)] < SYSIN[(NAMIN)]

POCONV POFILE < SYSIN[(NAMIN)] POTYPE [NR]

POLY

Creates and updates a scalar or matrix poynomial file

POLY [[AGOUT:]POLOUT] [<] [[AGIN:]POLIN] [NR NC] [TSAMP]

Subcommands:

```
LOOK [DEG]
KILL
X
INS [DEG]
INS [DEG] < VALUE
ALT VALUE [DEG] [NR NC]
ADDZ VRE [VIM]
MULC V
DIVC V
DEL [DEG]
```

POLZ

Computes the zeroes of a polynomial with real, scalar coefficients

```
POLZ [[AGOUT:]ZERFIL <] SYSIN[(NAMIN)] POTYPE [NR]

POLZ [[AGOUT:]ZERFIL <] [AGIN:POLY
```

PPLAC

Pole placement using state feedback in a dynamical single input system.

```
PPLAC L [ [SYST1][(NAME1)] ] < SYST2[(NAME2)] EVAL
```

RECON

Reconstruction of the state of a dynamical single output system using a Kalman filter.

```
RECON K [SYST1] < SYST2[(NAME2)] EVAL
```

REDUC

Generates a new matrix from a part of an old one

```
REDUC [[AG1:]M1] < [AG2:]M2 (IX1 IY1 IX2 IY2)
```

SAMP

Computes the system matrices for a sampled version of a continuous linear dynamic system

```
SAMP [SYSOUT][(NAMOUT)] < SYSIN[(NAMIN)]
```

SPSS

Computes the power spectrum or the amplitude and phase for the transfer function of a specified input-output pair of a discrete or continuous state-space model

```
SPSS [('POW'/'AMP')] FRF[(F)] < SYST[(NAME)] NY NU
                                        [FREQ]
```

SSTRF1

Transforms a Miso State Space model into a Transfer Function model

SSTRF1 [SYSOUT][(NAMOUT)] < SYSIN[(NAMIN)]

SYST

System description editor handling the following model types:

State Space Representation
Miso Transfer Function (polynomial file form only)
Polynomial Matrix Representation

SYST [(SUBSW)] SYSNAM[(SECNAM)] [< [(SYSTYP)] [SYSMNEM]
[DT] [AGRNAM] [(TIMTYP)/OP/LAMVAL] [ATRNAM]]

Subcommands:

BEGIN [SECNAM]
TSAMP DT
LOOK
AG [(AGTYP)] [AGRNAM] LAMBDA LAMVAL
INS MNEM [< NAME]
DEL MNEM
KILL
X

SYSTR

Transforms a dynamical system on a state space representation when the coordinates are transformed as Z=T*X

SYSTR [SYST1][(NAME1)] < SYST2[(NAME2)] T [EPS]

TBALAN

Transformation of a dynamical system in state space representation to get a balanced A-matrix.

TBALAN T [[SYST1][(NAME1)]] < SYST2[(NAME2)] [EPS]

TCON

Transformation to reachable canonical form of a dynamical single input system in state space representation.

TCON T [[SYST1][(NAME1)]] < SYST2[(NAME2)] [EPS]

TDIAG

Transformation to diagonal form of a dynamical system in state space representation.

TDIAG EIGVEC [[SYST1][(NAME1)]] < SYST2[(NAME2)]

THESS

Transformation to Hessenberg form of a dynamical system in state space representation.

THESS T [[SYST1][(NAME1)]] < SYST2[(NAME2)] [EPS]

TOBS

Transformation to observable canonical form of a dynamical single output system in state space representation.

TOBS T [[SYST1][(NAME1)]] < SYST2[(NAME2)] [EPS]

TRFSS1

Transforms a Miso Transfer Function model into a observable canonical State Space model

TRFSS1 [SYSOUT][(NAMOUT)] < SYSIN[(NAMIN)]

UNITM / ZEROM

Generates a zero or unit matrix

UNITM [* FACTOR] [AG:]MAT NR [TSAMP]

ZEROM [+ TERM] [AG:]MAT NR [NC] [TSAMP]

Synpac Command List

MATOP

Evaluates simple matrix expressions

MATOP [NAME1]<-UNIOP NAME2

MATOP [NAME1]<-NAME2 BINOP NAME3

SYST

Defines a system description

SYST SNAME<-NAME1 ... NAME4 [NAME5]

SAMP

Computes the discrete time representation of a system

SAMP [SNAM1<-SNAM2 [TSAMP]]

TRANS

Transforms a LQ problem to discrete time form

TRANS [LNAM1<-SNAME LNAM2 [TSAMP]]

STRIC

Solves the stationary Riccati equation

STRIC NAME<-SNAME LNAME

OPTFB

Computes the LQ optimal feedback

OPTFB NAME + SNAM1<-SNAM2 LNAME

REDFB

Computes the reduced feedback

REDFB [LRNAM<-SNAME LNAM FBS W]

KALFI

Computes the Kalman filter gain

KALFI KNAME<-SNAME RNAME

SYSOP

Computes the total system from its subsystems

SYSOP SNAMT<-SNAM1 [SNAM2 [..]..]

Subcommands are used to describe the connections to be made. Their exact syntax is quite complicated. Some simple examples are found in Chapter 8.

CORNO

Computes correlated noise

CORNO DATA<-R NP

SIMU

Simulates a discrete time state space system

SIMU [DNAM1[(C11 .. C1N)]<-SNAME DNAM2[(C21 .. C2M)]

EIGEN

Computes eigenvalues and eigenvectors of a matrix

EIGEN [[EVAL EVEC<-]NAME]

POLES

Computes poles and modes of a system

POLES [[EVAL EVEC<-]SNAME]

PRINT / TYPE

Prints or types a file on the line printer or the console device resp.

PRINT FNAME[(C1 C2 ..)] [IF NUM]

TYPE NAME